ROS2 Project Report: Publisher and Subscriber for Image Processing in C++

Dheeraj Bhurewar

14th October 2024

Abstract

This report details the creation of a ROS2 project in C++ involving the development of publisher and subscriber packages. The project demonstrates image processing using OpenCV integrated with ROS2, specifically for subscribing to an image topic, converting it to grayscale, performing contour detection, and publishing the processed image. The report also covers handling various edge cases, testing, findings, and recommendations for improving inference time.

Contents

Chapter 1

Introduction

1.1 Overview

This project involves creating ROS2 packages in C++ to demonstrate the publisher-subscriber model for image processing. The goal was to subscribe to an image topic, process the image by converting it to grayscale, detect contours, and publish the processed image. The project utilized OpenCV for image processing tasks.

1.2 Objectives

- Develop a ROS2 subscriber node that receives images from a topic.
- Convert the incoming image from RGB to grayscale.
- Perform contour detection on the grayscale image.
- Publish the processed image.
- Handle various edge cases during image processing.

Chapter 2

System Specifications

2.1 Hardware Specifications

- **Laptop Model:** Lenovo Legion 5 Pro
- **CPU:** AMD Ryzen 7 5800H, 8 cores, 16 threads, 3.2 GHz base clock, 4.4 GHz max boost clock
- **GPU:** NVIDIA GeForce RTX 3060
- **RAM:** 16 GB DDR4
- **Storage:** 512 GB SSD

2.2 Software Specifications

- **Operating System:** Ubuntu 22.04 LTS
- **ROS2 Distribution:** ROS2 Humble
- **Programming Language:** C++
- **Dependencies:** OpenCV, cv_bridge, rclcpp, image transport

Chapter 3

Methodology

3.1 Step-by-Step Guide

The methodology involves creating a ROS2 project for image processing, following these steps:

3.1.1 1. Setting up the ROS2 Environment

To ensure ROS2 Humble is installed and correctly set up:

```bash
# Source the ROS2 installation
source /opt/ros/humble/setup.bash

# Create a new workspace for the project
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/
```

3.1.2 2. Creating ROS2 Packages

Create two packages: one for the image publisher and one for the image subscriber.

```bash
# Navigate to the src folder within the workspace
cd ~/ros2_ws/src

# Create the image_publisher package
ros2 pkg create --build-type ament_cmake ros88_pub

# Create the image_processor package for the subscriber
ros2 pkg create --build-type ament_cmake ros88
```

3.1.3 Writing the Publisher and Subscriber Code

The next step involves implementing the publisher and subscriber nodes.

- **Publisher Node:** This node simulates a camera by publishing images to a topic.
- **Subscriber Node:** This node subscribes to the image topic, processes the image, and republishes it after performing operations like grayscale conversion and contour detection.

3.1.4 Modifying CMakeLists.txt and package.xml

Update the `CMakeLists.txt` and `package.xml` files in both packages to include the necessary dependencies.

```
# In CMakeLists.txt, add the following lines to find OpenCV
    and rclcpp
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(cv_bridge REQUIRED)
find_package(OpenCV REQUIRED)

# Link libraries for the executable nodes
target_link_libraries(<node_name> ${OpenCV_LIBS})
```

Add dependencies in the `package.xml` file:

```
<depend>rclcpp</depend>
<depend>sensor_msgs</depend>
<depend>cv_bridge</depend>
<depend>OpenCV</depend>
```

3.1.5 Building the Workspace

To build the packages, run the following commands:

```
# Navigate to the root of the workspace
cd ~/ros2_ws/

# Build the workspace using colcon
colcon build

# Source the workspace
source install/setup.bash
```

3.1.6 Launching the Nodes

Use the ROS2 CLI to run the publisher and subscriber nodes.

```
# Run the image publisher node
ros2 launch ros88_pub ros_pub_launch.py

# Run the image processor (subscriber) node
ros2 launch ros88 ros_sub_launch.py
```

3.1.7 Testing and Verification

Test the implementation by publishing different types of images (various resolutions, noise levels, and content). Use ROS2 commands to check the output:

```
# Publish a test image to the input topic
ros2 topic pub /my_camera/image sensor_msgs/Image '<image
    data>'

# View the output topic to confirm processing
ros2 run rviz2 rviz2
```

3.1.8 Handling Edge Cases

Specific code was implemented to manage various edge cases:

- **Resizing Images:** The images were resized to a standard 640x480 resolution using OpenCV's `resize` function, no matter if the size were 3840x2160, 1920x1080, 960x540, or even 480x640.
- **Noisy Images:** Applied a Gaussian blur with the `cv::GaussianBlur` function to smooth noisy images before processing.
- **Corrupted Images:** Added try-catch blocks to handle exceptions thrown by OpenCV functions.

Example command for testing with corrupted images:

```
# Corrupt an image by truncating it
head -c 1000 original.jpg > truncated.jpg
```

3.2 ROS2 Commands Summary

- **Creating a package:** `ros2 pkg create --build-type ament_cmake <package_name>`
- **Building the workspace:** `colcon build`
- **Running a node:** `ros2 launch <package_name> <node_executable>`
- **Publishing a topic:** `ros2 topic pub <topic> <msg_type> <data>`
- **Viewing a topic:** `ros2 run rviz2 rviz2`

Chapter 4

Test Cases

This chapter outlines the test cases used to evaluate the functionality of the image processing system. Each subpoint includes specific code snippets from the subscriber node's implementation, demonstrating how different cases were handled.

4.1 Handling Different Image Resolutions

The subscriber node was designed to resize incoming images to a standard resolution (640x480) for consistent processing. The code snippet below shows how the resizing was performed using OpenCV's `resize` function:

```
// Resize the input image to 640x480 resolution
cv::Mat resized_image;
cv::resize(input_image, resized_image, cv::Size(640, 480));
```

Test cases included images with resolutions:

- 3840x2160 (4K)
- 1920x1080 (Full HD)
- 960x540 (QHD)
- 480x640 (Vertical orientation)

4.2 Handling Noisy Images

To improve processing accuracy, a Gaussian blur filter was applied to noisy images before processing. The following code snippet shows how the blur was applied:

```cpp
// Apply Gaussian blur to reduce noise
cv::Mat blurred_image;
cv::GaussianBlur(resized_image, blurred_image, cv::Size(5, 5)
    , 0);
```

Testing was conducted using images with different levels of random noise added. Results showed improved contour detection after noise reduction.

4.3 Images with No Contours

In cases where the image did not contain any detectable contours, the system was designed to log a message and skip the processing step. The following code demonstrates this handling:

```cpp
// Find contours in the processed image
std::vector<std::vector<cv::Point>> contours;
cv::findContours(edge_image, contours, cv::RETR_EXTERNAL, cv
    ::CHAIN_APPROX_SIMPLE);

// Check if any contours were found
if (contours.empty()) {
    RCLCPP_INFO(this->get_logger(), "No contours found in the
     image.");
    return; // Exit processing
}
```

For these test cases, images without clear edges (e.g., a blank white image) were used, and the system appropriately handled the scenario by logging the absence of contours.

4.4 Handling Multiple Contours

When an image contained multiple contours, the subscriber node processed each contour individually and drew them on the image. Here's how it was implemented:

```cpp
// Iterate over all found contours and draw them
for (const auto& contour : contours) {
    cv::drawContours(output_image, contours, -1, cv::Scalar
   (0, 255, 0), 2);
}
```

Test cases involved images with shapes such as triangles, rectangles, and circles to verify that multiple contours were correctly identified and drawn.

4.5 Corrupted Images Handling

Corrupted images or improperly formatted data can cause exceptions during processing. The system catches exceptions using a try-catch block to handle these cases:

```cpp
try {
    // Attempt to process the image
    cv::Mat corrupted_image = cv::imdecode(buffer, cv::IMREAD_COLOR);
    if (corrupted_image.empty()) {
        throw std::runtime_error("Corrupted or invalid image data.");
    }
} catch (const std::exception& e) {
    RCLCPP_ERROR(this->get_logger(), "Failed to process the image: %s", e.what());
}
```

Testing was conducted using images corrupted by truncation or tampering. For instance, running the command `head -c 1000 original.jpg > truncated.jpg` generated a corrupted image. The system successfully caught the exception and logged the error.

4.6 Inference Time Analysis

The inference time for image processing was calculated by measuring the duration between the receipt of the image and the completion of contour detection. The following code shows how this was achieved:

```cpp
// Record the start time
auto start_time = std::chrono::steady_clock::now();

// Perform the image processing tasks here...

// Record the end time
auto end_time = std::chrono::steady_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();
RCLCPP_INFO(this->get_logger(), "Inference time: %ld ms", duration);
```

Test cases included measuring the inference time for images of different sizes and levels of complexity. Offline processing was used, but for online processing (e.g., video streams), the system could potentially experience FPS (frames per second) drops depending on the computational load.

4.7 Handling Situations with No Published Images

In cases where no image was being published on the input topic, the system simply waited without performing any operations. No exceptions were thrown since the subscriber did not receive any data to process. This was implicitly handled by the ROS2 messaging system's callback mechanism.

4.8 Failure Cases

Specific failure cases such as corrupted data, unrecognized formats, or hardware issues were simulated to observe the system's behavior. For example:

- **Unrecognized Image Format:** Attempting to process non-image data resulted in an appropriate exception message.

4.9 Findings

- **Inference Time:** The average processing time was 5-6 ms per image.
- **Failure Cases:** Corrupted images were correctly identified.

Chapter 5

Failure Handling

5.1 Scenarios

- **No Image Being Published:** If the input topic is inactive, the node remains idle without generating errors.
- **No Contours Found:** The original image is published in gray scale.
- **Corrupted Images:** Corrupted or unreadable images were skipped, and an error message was logged.

Chapter 6

Inference Time Considerations

- **Offline Processing:** In offline mode, the current implementation achieves adequate processing speed.
- **Online Processing (Video Streams):** Processing real-time video streams requires optimized algorithms and hardware acceleration (e.g., using GPUs) to maintain higher frame rates. For video with 30 FPS, the current processing time would introduce latency, necessitating further optimizations.

Chapter 7

Results

This chapter presents the results of the contour detection system on various types of images. The images demonstrate how the system performs under different conditions, such as low light, complex backgrounds, or images with multiple objects. The results are labeled to indicate the type of scenario being tested.

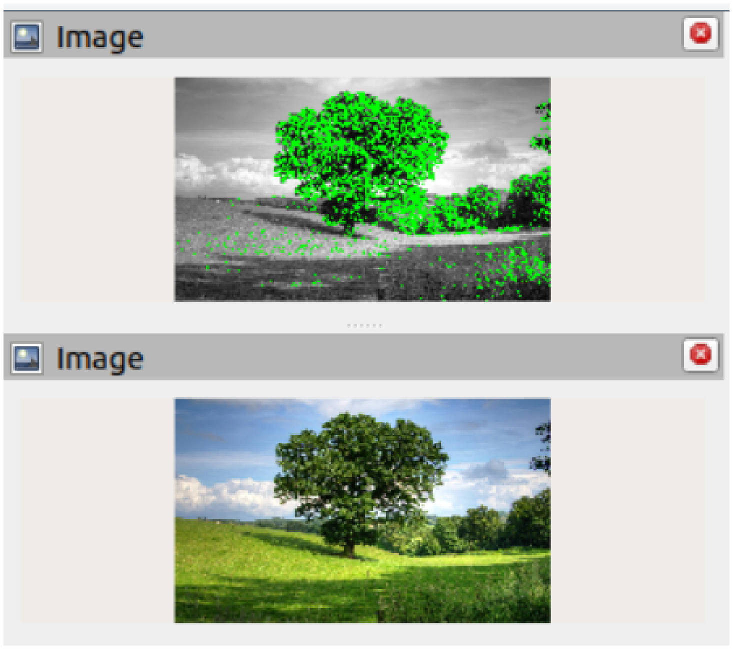

Figure 7.1: Contours detected in an image of trees.

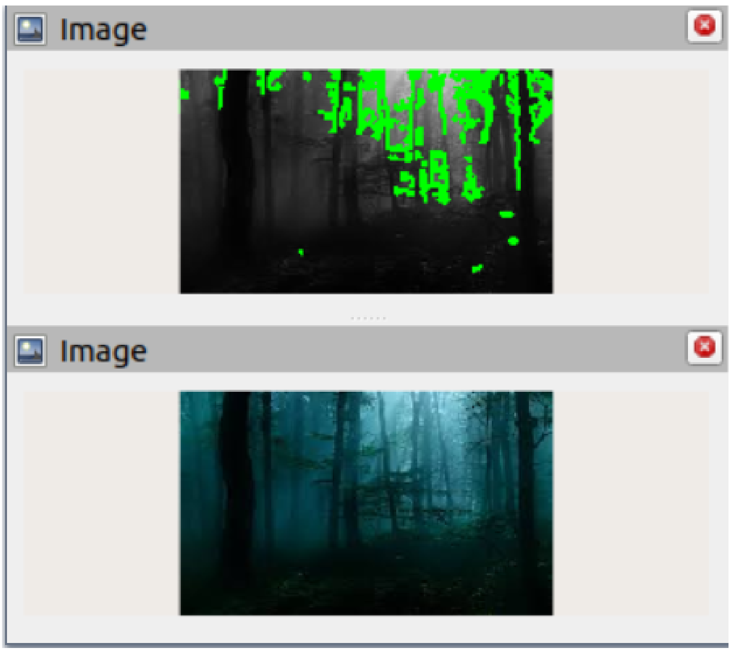

Figure 7.2: Contour detection results for a low-light image.

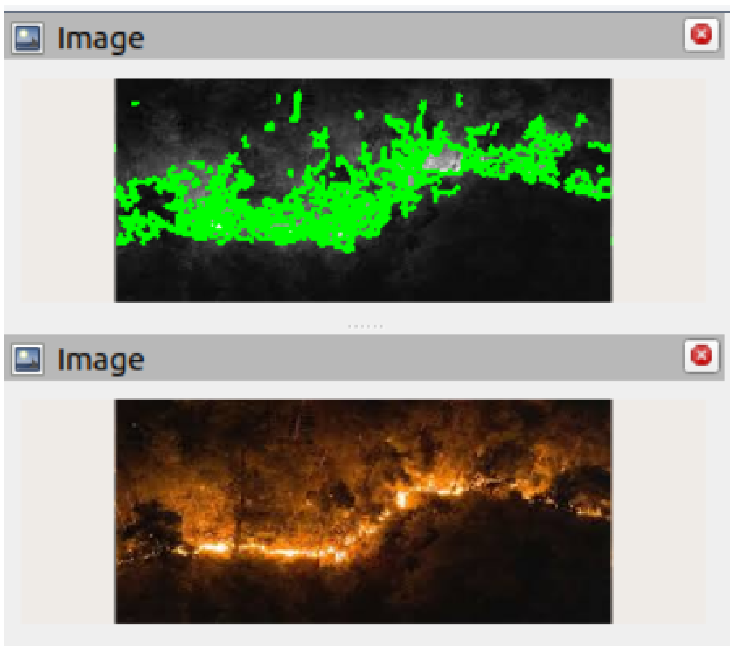

Figure 7.3: Contours detected in an image of a wildfire.

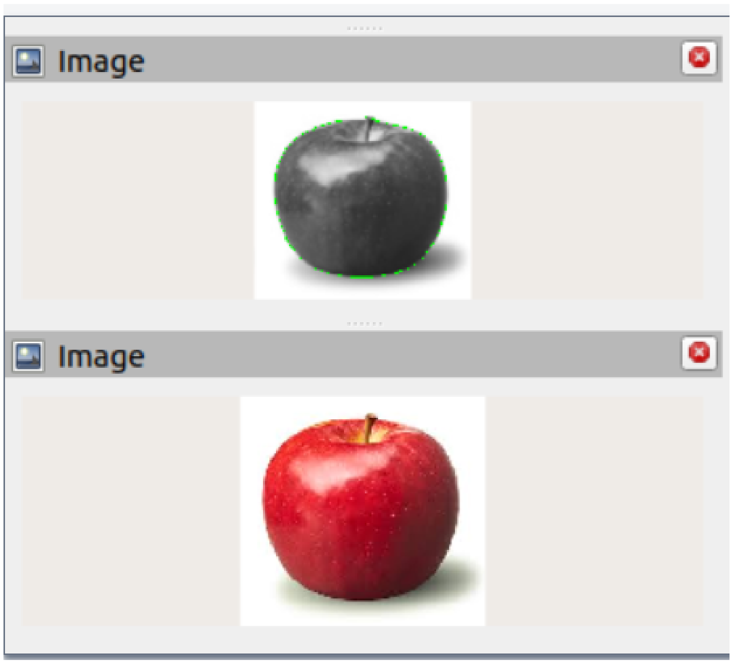

Figure 7.4: Detected contours for an image with a single object (e.g., a ball).

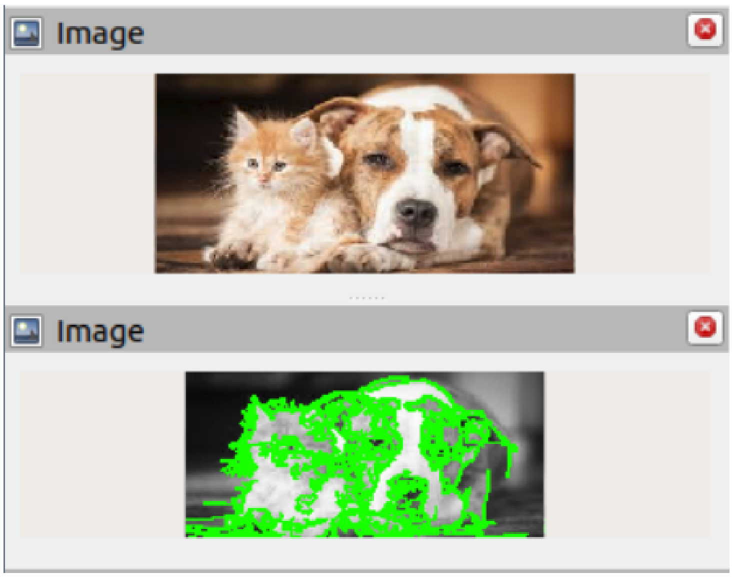

Figure 7.5: Contour detection results for an image with multiple living creatures.

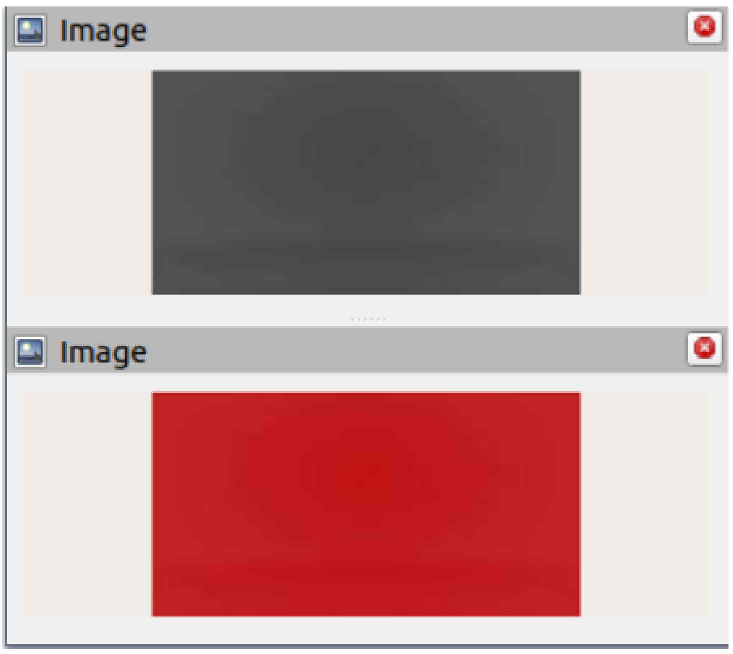

Figure 7.6: An image with no detectable contours.

Chapter 8

Evaluation

The results obtained from these test cases validate the system's functionality across a diverse set of conditions, while also highlighting areas where further enhancements could be made. The solution handles different edge cases well. Further work could involve optimizing inference times for real-time applications.

Chapter 9

References

- ROS2 Documentation: `https://docs.ros.org/en/humble/index.html`
- OpenCV Documentation: `https://docs.opencv.org/`