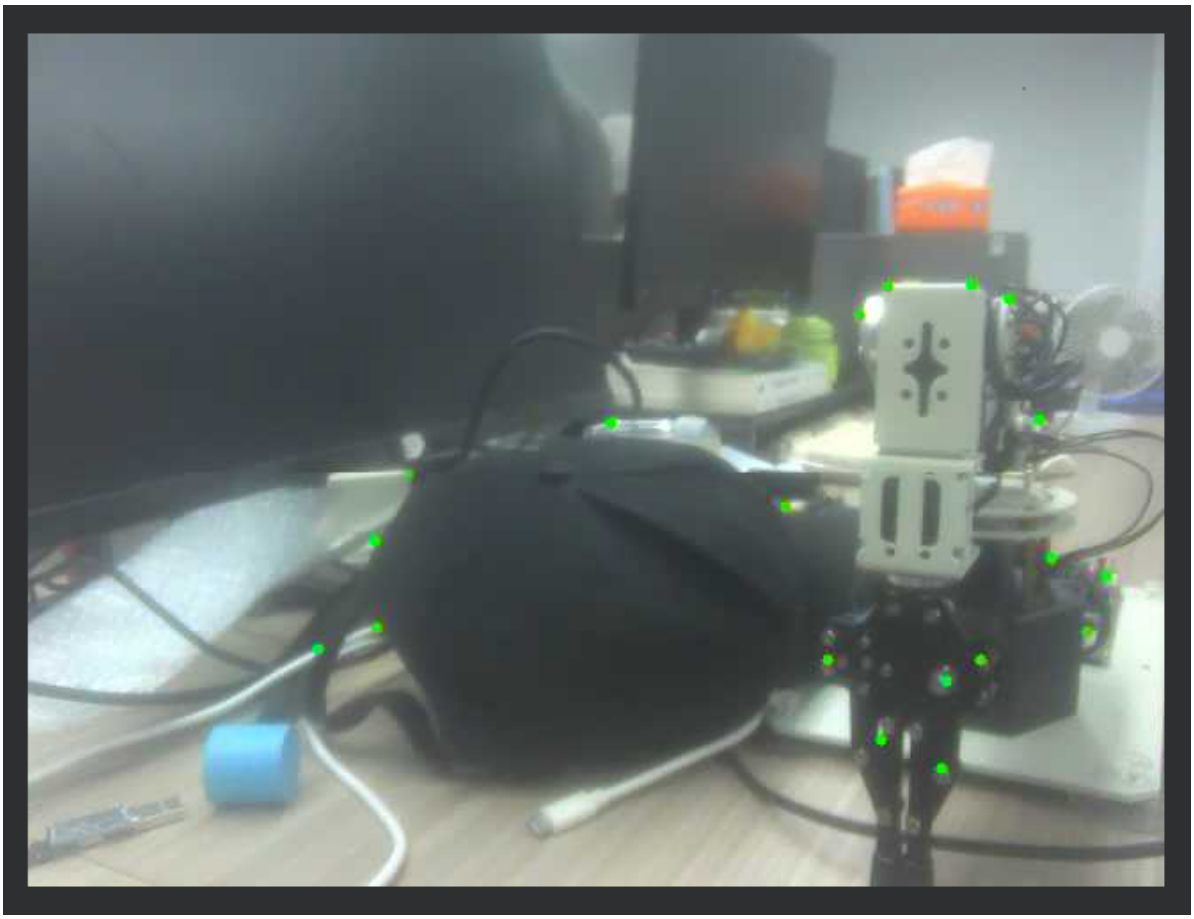
Corner Recognition - Color Image

Example Results

Run this section's example code [Source Code/06.cv_lite/30.rgb888_find_corners.py]

In this section, we'll use the `cv_lite` extension module to implement RGB888 image corner detection on an embedded device. We'll detect and display key corners in the image for feature extraction and object tracking.



Principle Explanation

What is RGB888 Image Corner Detection?

RGB888 image corner detection is a computer vision technique used to identify points (i.e., corners) in color images (in RGB888 format) with dramatic brightness or color changes. These points are typically located at the edges or corners of objects and are highly distinctive. Corner detection is based on grayscale conversion or direct processing of color images. Common algorithms, such as the Shi-Tomasi corner detector, can efficiently extract stable keypoints.

- **Operational Principle**: Corner detection algorithms first convert an RGB888 image to grayscale or directly calculate the gradient matrix (Hessian matrix) in color space. They then evaluate the corner response value of each pixel (based on the eigenvalues of a local window). High-quality corners are selected by setting a quality threshold and a minimum distance. The formula involves calculating the eigenvalues λ1 and λ2 of a pixel. If min(λ1, λ2) > the threshold, the pixel is considered a corner. This process is performed on RGB888 images to enhance detection accuracy by leveraging color information.
- **Effect**: The detection results highlight keypoints in the image and mark them, improving the accuracy of visual analysis. After processing, corners can be used for subsequent tasks such as object tracking or matching, but detection accuracy is affected by image noise, color distortion, and parameters.
- **Application Scenarios**: RGB888 corner detection is widely used in real-time object tracking (such as visual SLAM), motion detection, image registration, and robotic vision. On embedded devices, its high computational efficiency makes it suitable for resource-constrained environments, such as cameras processing color images in real time.

In practical applications, by adjusting parameters, the robustness and number of detections can be optimized to accommodate varying image scenarios and color complexity.

Code Overview

Importing Modules

```
import time, os, sys, gc
from machine import Pin
from media.sensor import * # Camera interface
from media.display import * # Display interface
from media.media import * # Media manager
import _thread
import cv_lite # cv_lite extension module
import ulab.numpy as np
```

Setting the Image Size

```
image_shape = [480, 640] # Height x Width
```

Define the image resolution to 480x640 (height x width). The camera and display modules will be initialized based on this size.

Initialize the camera (RGB888 format)

```
sensor = Sensor(id=2, width=1280, height=960,fps=60)
sensor.reset()
sensor.set_framesize(width=image_shape[1], height=image_shape[0])
sensor.set_pixformat(Sensor.RGB888)
```

- Initialize the camera, set the resolution to 1280x960 and the frame rate to 60 FPS.
- Resize the output frame to 640x480 and set it to RGB888 format (three-channel color image, suitable for corner detection to utilize color information).

Initialize the display module

```
Display.init(Display.VIRT, width=image_shape[1], height=image_shape[0],
to_ide=True, quality=50)
```

Initialize the display module, use the virtual display driver (Display.VIRT), and set the resolution to match the image size. `to_ide=True` indicates that the image will be transferred to the IDE for virtual display at the same time, and `quality=50` sets the image transfer quality.

Initialize the media manager and start the camera

```
MediaManager.init()
sensor.run()
```

Initialize the media resource manager and start the camera to capture the image stream.

Set the frame rate timer

```
clock = time.clock() # Start the frame rate timer / Start FPS timer
```

Initialize the frame rate timer, which is used to calculate and display the number of frames processed per second (FPS).

Set corner detection parameters

```
max_corners = 20 # Maximum number of corners
quality_level = 0.01 # Corner quality factor (0.01 to 0.1)
min_distance = 20.0 # Minimum distance between corners
```

- `max_corners`: Maximum number of detected corners, limiting the output results.
- `quality_level`: Corner quality threshold, used to filter out corners with high response values.
- `min_distance`: Minimum distance between corners, to avoid detecting too dense points.

Image Processing and Corner Detection

```
while True:
    clock.tick()

    # Capture current frame
    img = sensor.snapshot()
    img_np = img.to_numpy_ref() # Get RGB888 ndarray reference
```

```
    # Call corner detection function, return corner array [x0, y0, x1, y1, ...]
    corners = cv_lite.rgb888_find_corners(
        image_shape, img_np,
        max_corners,
        quality_level,
        min_distance
    )
    # Traverse corner array and draw detected corners
    for i in range(0, len(corners), 2):
        x = corners[i]
        y = corners[i + 1]
        img.draw_circle(x, y, 3, color=(0, 255, 0), fill=True)


    # Display image with corners
    Display.show_image(img)

    # Perform garbage collection
    gc.collect()

    # Print current FPS and number of corners
    print("fps:", clock.fps(), "| corners:", len(corners) // 2)
```

- **Image Capture**: Acquire an image frame using `sensor.snapshot()` and convert it to a NumPy array reference using `to_numpy_ref()`.
- **Corner Detection**: Call `cv_lite.rgb888_find_corners()` to perform corner detection and return an array of corner coordinates.
- **Drawing and Display**: Iterate over the detected corners, draw green circle markers on the image, and display them on the screen or in an IDE virtual window.
- **Memory management and frame rate output**: Call `gc.collect()` to clear memory, and print the current frame rate and number of corner points through `clock.fps()`.

Resource release

```
sensor.stop()
Display.deinit()
os.exitpoint(os.EXITPOINT_ENABLE_SLEEP)
time.sleep_ms(100)
MediaManager.deinit()
```

Parameter adjustment instructions

- `max_corners`: Maximum number of corner points. A larger value (e.g., 10-50) detects more corner points, but may increase the computational burden. A smaller value (e.g., 5) reduces the number of points to increase speed. It is recommended to start testing with 20 based on image complexity.
- `quality_level`: Shi-Tomasi quality factor. Higher values (e.g., 0.01-0.1) require stronger corner response, resulting in more accurate detection but potentially fewer corners. A value too low (e.g., 0.001) may detect noise. It is recommended to start fine-tuning from 0.01.
- `min_distance`: Minimum corner distance. Higher values (e.g., 10-50) ensure corner spacing and avoid densely packed corners. Lower values (e.g., 5) result in denser detection but may include redundancy. It is recommended to adjust this value based on image resolution and target scene. A starting value of 20 is recommended for testing.