Software Report

smart_cap/stereoscopic/capture_calibration.py

This module handles capturing images for calibrating the stereoscopic camera setup. The only necessary inputs for this module are the camera feeds from both cameras. It is set up to work with two 1920 by 1080 USB cameras. Based on how the cameras are plugged in, they will be mapped to device number 2 for the left camera and device number 0 for the right camera. These parameters are the only things that should be subject to change if the cameras and their connection scheme are changed. The module saves the captured images to directories called left and right. When capturing calibration images for a new stereo camera setup, it is necessary to empty the contents of output directories first. It may be helpful to empty the output directories even when using the same camera set up in the event that an undesirable calibration error is obtained. It can also be useful to capture additional images without overwriting those in the output directories. This can be done by changing the `image_count` variable to a number higher than the greatest number in the output directory. To operate the script, both cameras must be plugged in. When the script is running, it will show a live feed of the camera outputs. To generate the calibration images, all one needs to do is place the calibration reference image in the frames of both cameras and press `c` to capture images. When a sufficient number of images are captured, `q` can be pressed to quit out of the script.

smart_cap/stereoscopic/calibration.py

This module performs stereoscopic camera calibration given a calibration data set of images captured of a 10 by 7 checkerboard pattern of 25 millimeter squares. It then performs stereoscopic calibration by identifying the checkerboard pattern in each pair of images from the left and right cameras and finding the disparities between common points. This is then saved to a file called `stereo_calibration.npz` which can be renamed and referenced in the future.

smart_cap/pi.py

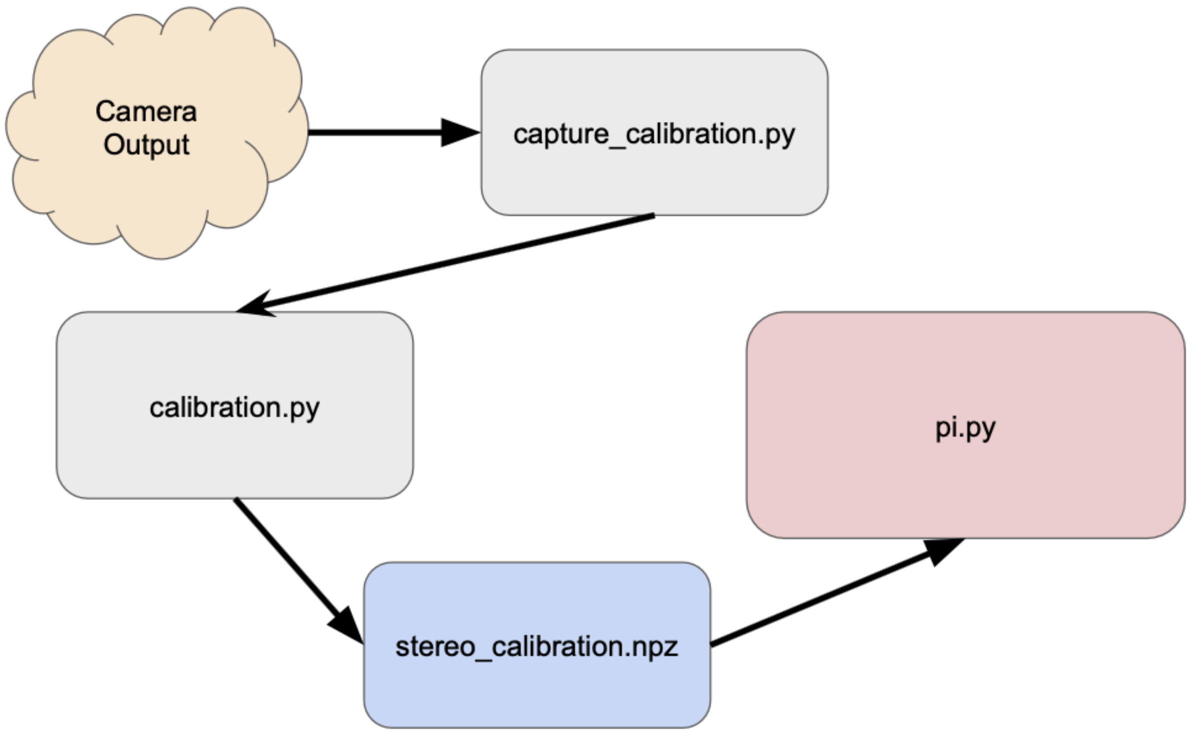
This module runs all the code needed for our Raspberry Pi. It first opens up a UDP server to allow for communication with the ESP32. It then starts initialization of the cameras, text-to-speech module, and our cloud-based LLM. The camera initialization includes the calibration matrix obtained from the calibration modules. For our LLM initialization, we created a prompt to feed our LLM. Next, the code waits for a response from the UDP server, which will be a button press from the wrist-wearable. This will start up the whole Raspberry Pi process, which includes capturing an image using the two cameras, mapping it with conversion based on the calibration settings, and running the Gemini LLM. Our LLM inputs include the calibrated images and API key, and our output is text response, as well as a boolean 'SAFE' or UNSAFE'.

This boolean is sent to the ESP32 through the UDP server, and the text response is sent to the headset’s speakers through a text-to-speech module.

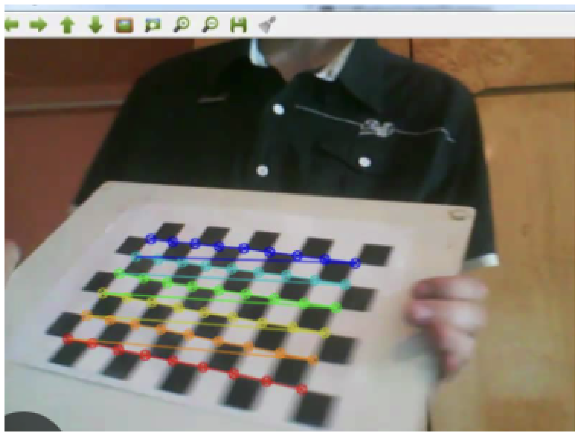
wrist_wearable/esp.py

This module implements all core functionality for a wrist-wearable device using an ESP32 microcontroller. It establishes a WiFi connection to a specified network and communicates with a UDP server to send and receive real-time messages. The device continuously monitors ambient light using a photoresistor, dynamically adjusting the brightness of an onboard LED with a smooth, sine-wave gradient when low light conditions are detected. This ensures the device remains visible and responsive to changes in the surrounding environment.

User interaction is facilitated through a physical button. When the button is pressed, the module sends a notification to the server and awaits a response. Depending on the server’s reply, the device activates a vibration motor to provide haptic feedback: a single pulse for standard acknowledgments or a series of strong pulses for unsafe or emergency alerts. The ESP grabs either a ‘SAFE’ or ‘UNSAFE’ input from the Raspberry Pi which determines the vibration motor’s strength. The module is designed with efficient state management to avoid repeated signals and ensures all subsystems-light sensing, button detection, vibration feedback, and network communication-operate concurrently and responsively within the main loop. This makes the module suitable for applications where immediate feedback and environmental awareness are critical, such as personal safety or alert systems.



Flow chart indicating dependencies between functions, programs, and hardware outputs. Initially the cameras output multiple photographs that are used for calibration. **capture_calibration.py** handles all of the photographs that are taken and organizes them into two separate buckets, one for the left camera and one for the right camera. For optimal calibration, we recommend taking 20 photos on each camera; the **capture_calibration.py** script handles capturing photos for both cameras at the same time. Be sure to hold still when taking photos as there is a small delay in between taking the left and right camera photo. Photos taken with the capture_calibration.py should be of chessboard black and white patterns. See image below for reference. The chessboard image can be printed or displayed using a tablet. Multiple angles of the chessboard should be taken.



Once all of the photos are taken, **calibration.py** will then begin by detecting chessboard patterns in the image pairs. The script uses OpenCV's `findChessboardCorners` to refine the black and white intersections in both left and right camera images. For each successful detection, it refines the corner coordinates with sub-pixel accuracy using `cornerSubPix`, then it stores the 3D object points (real-world chessboard coordinates) and 2D image points from both cameras.

Individual camera calibration follows, where calibrateCamera calculates each camera's intrinsic parameters:

- Camera matrix (focal length and optical center)
- Distortion coefficients (radial and tangential distortion)
- RMS reprojection error (lower values and indicate better calibration)

The stereo calibration phase (`stereoCalibrate`) then determines the spatial relationship between cameras by solving for:

- Rotation matrix (R) and translation vector (T) between cameras
- Essential matrix and fundamental matrix for epipolar geometry
- Refined camera matrices accounting for stereo alignment

Finally, all calibration parameters get saved to `stereo_calibration.npz` for use in downstream applications like depth mapping. The script outputs RMS errors at each stage, with typical successful calibrations achieving errors below 0.5 pixels.

Dev/Build Tool Information

The Raspberry Pi is running Python 3.11.2. No other build tools or languages are used. The core libraries and their versions are as follows.

google-genai==1.9.0
Pillow==9.4.0
opencv-python==4.11.0.86
pyttsx3==2.98
numpy==1.24.2
python-dotenv==0.21.0

The TinyPICO ESP32 is running MicroPython v1.24.1. Only libraries from the standard library are used, and no other build tools or languages are used.

Installation Guide

The TinyPICO ESP32 comes with the appropriate MicroPython version installed. Since no other libraries are required, the project can be considered fully installed and functional once the GitHub repository is cloned. If another MicroPython-compatible device were to be used in future iterations instead of the TinyPICO ESP32, then that would be the only other dependency that would need to be installed. This can be done easily using Thonny IDE.

The Raspberry Pi does not have many dependencies to install. First, you must install Python 3.11.2. With Python installed, a developer can create a virtual environment and install the dependencies by running the following: `pip install -r requirements.txt`.

Lastly, a `.env` file must be created with the contents `GOOGLE_API_KEY=...`, replacing the ellipses with the actual contents of a valid Google AI Studio API key. API keys can be generated from Google Studio at the following link: https://aistudio.google.com/app/apikey. The rate limits for free users are generous enough to serve developers.