Lab Course: Hardware/Software Co-Design with a LEGO Car

SS 16

Abdallah Attawia, Moritz Dötterl, Heiko Lengenfelder, Berkay Sümer
Technische Universität München, Garching, Department of Informatics
Boltzmannstr. 3, 85748 Garching, Germany

This document explains the work done on a lego car to detect and follow a line autonomously. Connection and communication between the hardware parts is explained. Furthermore, collision avoidance and line detection algorithms are thoroughly described.

Contents

I Introduction 2

II Components overview 2
- II.A DEO-Nano . . . 2
- II.B Raspberry Pi . . . 2
- II.C Actuators . . . 3
  - II.C.1 Speed motors . . . 3
  - II.C.2 Servo motor . . . 3
- II.D Sensors . . . 3
  - II.D.1 Ultrasound sensor . . . 3
  - II.D.2 USB Camera . . . 3

III Hardware 3
- III.A Components . . . 3
- III.B Power Supply . . . 4
  - III.B.1 Power Connection . . . 4
- III.C Sensors . . . 4
  - III.C.1 Sensor Connection . . . 6
- III.D Control . . . 6
  - III.D.1 Control Connection . . . 6
- III.E Motors . . . 6
  - III.E.1 Motors Connection . . . 6

IV Collision Avoidance 7

V Image processing for line detection and following 7
- V.A Code . . . 7
  - V.A.1 Organising the threads . . . 8
  - V.A.2 Line detection algorithm . . . 8
  - V.A.3 Parameters . . . 8
  - V.A.4 Sequential Improvements 9
  - V.A.5 Exploit Parallelism . . . 10

VI Conclusion 12

I Introduction

An autonomous car is one that can drive from point A to B independently without any human input. It can accelerate, brake and steer itself as well as sense the environment and navigate to avoid crashing. The development of autonomous vehicles is rapidly growing and a lot of big and small companies are working towards making autonomous driving a reality.
In this project a small scaled model, a lego car, along with some cheap components are used to test a real autonomous driving function, i.e. following a line on the street. This documentation starts with a brief explanation of the different hardware components. Then the cabling and communication between the components is described. Lastly, the collision avoidance technique and and the line detection and following are explained. The code, attached to this report, carries out the functions explained here and is clearly documented.

II Components overview

The hardware components used are two controllers, the DEO-Nano and the Raspberry Pi, two types of actuators, namely speed motors and servo motors, as well as two types of sensors, which are the ultrasound sensors and the USB-camera.

II.A DEO-Nano

The DE0-Nano board is a compact-sized FPGA based development platform manufactured by terasIC. Its heart is a Cyclone IV FPGA from Altera. Furthermore it features on board memory, LEDs, Buttons, and most importantly three GPIO headers. On this FPGA we flashed the provided Nios-II image. This is an image of an adaptable processor allowing for custom specializations such as multiple hardware UART handlers. Now it is possible to wright code for this processor in C using the hardware you desire, without any trade off. We are using four UART units to connect to three ultrasonic sensors as well as to the Raspberry Pi. Furthermore we use two PWM modules with each two channels allowing us to generate PWM signals at four pins. Those are used to control the drive motor and the steering servo.

II.B Raspberry Pi

For this project the newly released Raspberry Pi 3 is used. Raspberry Pi is a small-sized one board computer. In this work it is used to connect the camera with the Nano-board and to run the line following code.
First, the micro SD-card for the Pi is prepared with the Raspbian Operating System installed on it. Then the Pi is boots the Raspbian and the computer is connected to the Raspberry Pi using Secure Schell (SSH) either via the Ethernet port, or using the build in Wi-Fi of the Pi 3 and the LRZ Wi-Fi network. Then the camera is connected using one of the USB ports and OpenCV is installed to run the line following code. Finally UART communication is enabled to connect the Raspberry Pi with the Nano-board. In Raspberry Pi 3 you need to disable Bluetooth in order for UART communication to work. To disable onboard Pi3 Bluetooth and restore UART over GPIOs 14 and 15, modify the file "/boot/config.txt":

- sudo nano /boot/config.txt
- Add to the end of the file: "dtoverlay=pi3-disable-bt"

II.C Actuators

II.C.1 Speed motors

The car has two Lego based driving motors that control the speed of the car. The motors are wired together to act as one motor, but with double the power. One motor drives the front tires, the other motor the rear tires to provide maximum traction.

II.C.2 Servo motor

The car uses one Lego servo motor to steer the direction. The servo steers both axles of the car to provide maximum agility and a relatively low turning cycle.

II.D Sensors

II.D.1 Ultrasound sensor

The ultrasound sensor measures the distance from any detected object within its range. As shown in Figure 1 the ultrasound sensors detects objects by sending out ultrasound waves and receiving an echo, if an object is detected. The sensor measures the time of flight between transmitting and receiving the signal and calculates the distance according to the following formula (1).

$$L = \frac{340(m/s)x \triangle t(s)}{2} \quad (1)$$

In this course, the KS103 ultrasound sensor is used. It has a range up to 11m which can be specified by sending a command from the Nano-board. This sensor can operate using two digital communication protocols; I2C and Uart. It has an accuracy between $1mm$ and $10mm$
In order for the ultrasound sensor to operate, it is first initialized and set to the desired range.

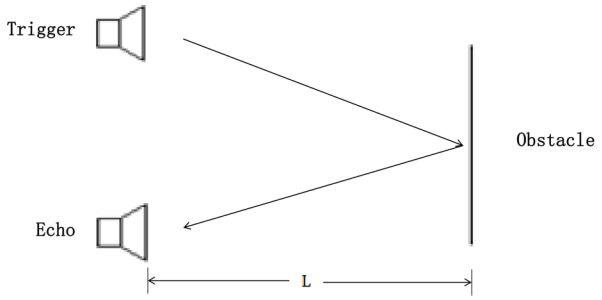

Figure 1: Theory of operation of the ultrasound sensor

Like the raspberry pi, it communicates with the Nano-board using uart communication, which has to be initialized as well. The Nano-board then sends a command to the sensor to send the data using uat. The Nano-board finally receives date from the sensor. The data read from the sensor is multiplied times 170 to get the distance from detected objects in micrometer.

II.D.2 USB Camera

A standard USB web cam is used to provide a stream of images of the area in front of the car. How these images are processed is explained in chapter V.

III Hardware

we used several different devices to build up our Lego Car. This chapter explains what devices we have used, and how they are interconnected.

III.A Components

The Hardware can be divided into four sub groups listed below:

Power Supply

- Battery
- H-Bridge
- CPT 15W DC/DC Converter

Sensors

- KS103 Ultrasound Sensor
- Camera

Control

- DE0-Nano Board
- Raspberry Pi 3

Actuators

- Steering Servo Motor
- Drive Motors

Figure 2 shows the wiring diagram of the total car setup. Including the Ultrasound sensor, the Nano Board, the steering servo, the driving motor in the first row (from left to right) and the USB Camera, the Raspberry Pi, the DC/DC converter and the battery in the second row (also from left to right). The H-Bridge is shown between those two rows and the wiring is depicted as colored lines interconnecting all devices.

III.B Power Supply

As a power source, we are using a three cell LIPO battery called "eco-x 5000 Top Fuel" which provides 11.1V. An H-Bridge with integrated voltage divider is used to convert from battery voltage to 5V to power the Nano Board which works at 5V. Furthermore the H-Bridge is used to translate the 5V signals from the Nano Board to 11.1V signals for the Drive Motor. We are also using a DC/DC converter which is an electronic circuit that also converts from the battery voltage to 5V. It is used to power the Raspberry Pi, since the voltage divider of the H-Bridge is not powerful enough for the Raspberry Pi. To ensure all devices work together they are all connected to the common Ground of the battery.

III.B.1 Power Connection

The 11.1V red cable of the battery is connected with the VCC input of the H-Bridge, and with the red cable of the DC/DC converter. The black GND cable of the battery is connected with the GND input of the H-Bridge, and the black ground input of the DC/DC converter.
The 5V input of the Nano Board is connected to the 5V Output of the H-Bridge, alternatively it could also be wired to the 5V output of the DC/DC converter. The GND of the Nano board is connected to the Ground of the whole car, e.g at the GND input of the H-Bridge
The servo motor's 11.1V input is attached to the VCC input of the H-Bridge. GND input of the servo motor has to be connected with the common Ground, so the GND input of the H-Bridge.

III.C Sensors

A simple web cam and one "KS103" ultrasound sensor are used as sensors. In principle we got three ultrasound sensors, but we only needed one to get the distance to objects in front of the car. so this document will only focus on the single sensor.Based on this sensor the car will adopt it's speed. How this is done exactly is discussed in IV. The camera is used for the line detection which is discussed in chapter V.

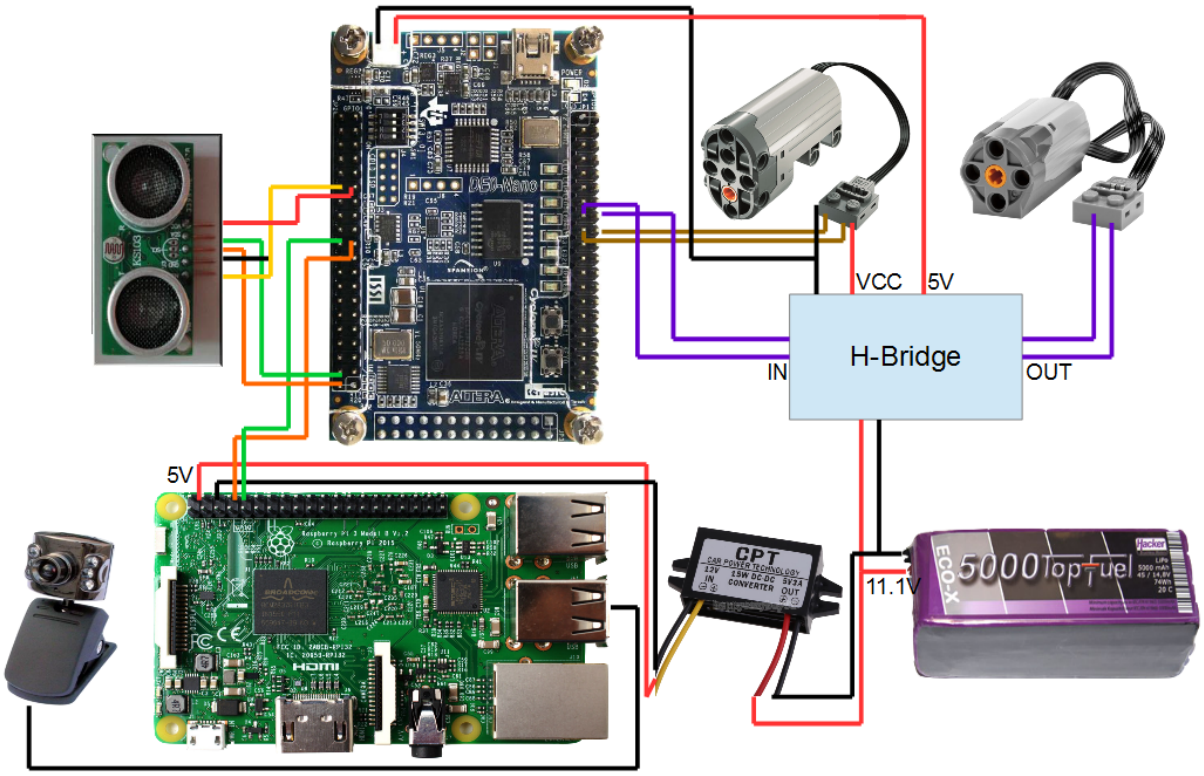

Figure 2: wiring diagram of the car

The KS103 Ultrasound Module has 5 pins (listed from right to left):

- VCC: power pin
- SDA/TX: data pin (I^2C) / Transmitter pin UART
- SCL/RX: clock pin (I^2C) / Receiver pin UART
- GND: power ground pin
- Mode: selects the communication mode (UART or I^2C)

There are two options for the communication: UART (universal asynchronous receiver/transmitter) and I^2C. Depending on the status of the Mode pin the sensor either expects I^2C (Mode pin high), or UART (Mode pin low) communication. We are using the UART modus in this project, which is an asynchronous serial communication format.

III.C.1 Sensor Connection

The 5V VCC and GND pins of the sensor are connected with the 5V and GND pins on the GPIO header of the Nano Board to power the sensor.
The Mode pin has to be connected with a GND pin of the Nano Board to select the UART mode. The TX and RX pins of the sensor are connected to the UART pins of the Nano Board. Note that, as usual with UART, the TX pin of the sensor is connected to the RX pin of the Nano Board and vice versa. GPIO_00 is connected with the TX pin of the sensor and GPIO_01 with the RX pin of the sensor.
The camera is simply plugged into one of the USB-ports of the Raspberry Pi.

III.D Control

We are using the "DE0-Nano Board" to provide an interface for the "Raspberry Pi 3" to control the car. The DE0-Nano Board introduces a compact-sized FPGA development platform suited for portable design projects, such as robots and mobile projects. A UART connection is used to enable communication between these two Boards. The calculation for the steering is done on the Raspberry Pi while the Nano Board generates the signals for the actuators and handles the connection to the ultrasonic sensor.

III.D.1 Control Connection

The UART pins of the Pi are connected with the UART pins of the Nano Board. The 8th pin of the Pi is the TX and 10th pin is the RX pin. They are connected to GPIO_016 and GPIO_017 of the Nano Board respectively.
The 2nd and 4th pins of the Pi are two 5V pins which can be used to power the Pi itself, or power other small devices from the Pi. Either one of them is connected with the the 5V output of the DC/DC converter. The 6th pin is the GND pin which has to be connected with the GND of the care e.g. the GND output of the DC/DC converter.

III.E Motors

We have two driving motors connected in a parallel manner and one servo motor on our car. The servo is used for the steering and the two driving motors are used to move the car. We are using PWM(Puls-width modulation) to generate analog control signals for the Motors. By using this modulation we had the ability to arrange the speed of our car and also the steering angle. The speed is adjusted by setting one pin of the driving motor to ground and generating a PWM signal on the other pin. The duty cycle of the PWM determines the speed of the car while switching the PWM and the ground pin causes the car to drive in the other direction. So setting one pin to '0' and the other pin to '1' would result in maximum speed while setting the pins to '1' and '0' would result in maximum reverse speed. The steering servo basically works the same way except that not the number of revolutions per time is controlled, but the angle of the servo. So setting the control pins to '0' and '1' respectively will cause the servo to steer all the way to one side, while setting the pins to '1' and '0' will make the servo steer all the way to the other side.

III.E.1 Motors Connection

The driving motors are both connected together, so they work as if we had one motor. The two pins of the motor are connected to the high volt-

age output of the H-Bridge. We are using OUT3 and OUT4 of the H-Bridge. The PWM output of the Nano Board is connected to the according low voltage input of the H-Bridge. We are using GPIO_112 and GPIO_113 on the Nano Board and connected them with IN3 and IN4 of the H-Bridge. Now the H-Bridge will transfer our signals to the 11.1V logic level the driving motor needs. Furthermore the high currencies needed by the motor will not flow through the Nano Board, which would be destroyed by them.
The two remaining cables of the servo motor are directly connected to the PWM output pins GPIO_114 and GPIO_115 of the Nano Board. Here no conversion between different logic levels is necessary and no high currencies could harm the Nano Board.

IV Collision Avoidance

In order to avoid collisions, an algorithm based on the idea of the adaptive cruise control is implemented. Adaptive cruise control is a cruise control function that allows the vehicle to adapt its speed according to the traffic and to maintain a safe distance from the vehicles ahead.
In real cars, a long range radar sensor is used to detect the cars, here we use an ultrasound sensor to measure the distance from the vehicles ahead. The following algorithm is implemented to avoid the collision with other cars or objects and to adapt the speed to the surrounding traffic.

- if distance from ultrasound sensor $\leq$ a minimum set distance ($distance_{min}$)
  -car stops
- if distance from ultrasound sensor $\geq$ a maximum set distance ($distance_{max}$)
  -drive with maximum speed
- else (distance lies within $distance_{min}$ and $distance_{max}$)
  -adapt speed according to distance:

$$CurSpeed = MaxSpeed * \frac{distance}{distance_{max}} \quad (2)$$

The speed of the car is set according to the aforementioned algorithm, where the current speed is the duty cycle that is assigned to the speed motors.

V Image processing for line detection and following

The task of the whole project was to build a Lego Car that can autonomous follow a line on the ground. For this purpose we are using an ordinary web cam on the front of the car giving us an image of the ground up to approximately 20 cm in front of the car. We developed an algorithm that detects the line in the image and calculates how the car needs to steer in order to stay on that line. The calculation is entirely done on the Raspberry Pi using the image processing framework OpenCV. The following sections explain how the image processing works and how we optimized it for the Raspberry Pi.

V.A Code

The source code that runs on the Raspberry Pi has two major subtasks to fullfill. The first one is to organise and utilize the output of the multiple threads that are used to do the image processing. The second one is to run the line detection algorithm on those threads.

V.A.1 Organising the threads

This task is done by the mainthread of the program. The communication between the mainthread and the computing threads is done via a synchronized variable. This variable represents the global direction the car should currently steer to. When a computing thread gets done with analysing a frame it updates this number. Before that it checks that this variable was not updated by a thread with a newer frame. Thus preventing old information being relevant. This global direction is not just set to the thread direction. Rather it is calculated by this formula:
$NewGlobalDir =$
$\alpha * ThreadDir + (1 - \alpha) * OldGlobalDir$
This is done to give the car more smooth driving and to counteract wrong direction caused by bad frames. The α-value is parameter that can be adjusted depending on the circumstances of the operating system and environment.
After a computing thread has written its data into this global variable it notifys the main thread. If the mainthread is not woken up it sleeps to prevent busy wait and sending data more often than necessary. When being called the mainthread takes this global direction and maps it 15 discrete steering angles. This is done because the steering servo only knows 15 settings. After doing so the mainthread sends the data via Uart connection to the Nano Board. This can be done by sending one byte because the values lie between zero and 14. When done with that the mainthread goes to sleep again.

V.A.2 Line detection algorithm

The algorithm is run on each thread separately to calculate the direction the car needs to take. A lot of steps are done using functions of the OpenCV library. This will be noted by writing:"(OpenCV)". To start of a thread grabs a frame of the camera (OpenCV). The frame needs to be blured to counteract noise on the picture. This is done via hardware. The picture from the camera uses RGB color coding. Because it is difficult to compute on three different channels the frame is being converted into the grey color scheme (OpenCV). After that the mask M is applied to the frame (OpenCV).

$$M = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

This should approximate the horizontal change of the picture. Because we assume the line is running vertically this should highlight the edges of it.
The algorithm now looks for the biggest value in each row. This should correspond to the right edge of the line if the line is the biggest contrast on the picture. Also it checks if this value is above a certain threshold and if two points are to far apart. If that is the case those points are not used in further computation. These thresholds can be varied depending on the camera position, amount of noise ect. Because of that the number of found points can vary for each frame. A line is now fit on these points (OpenCV). The cars position is assumed to be at bottom center of the picture. With the line and this assumption the needed angle to stay on the line can be calculated.
This angle is now used to update the global direction and the thread waits to grab a new frame.

V.A.3 Parameters

There are a several parameters in the source code that can be used to set the algorithm to fit in different environments.

ALPHA This corresponds to the influence of a newly calculated direction

STEERING_LEVEL_SIZE Range of non discrete steering angle

STEERING_LEVEL_SIZE_PROGRESSION Used if angle mapping should not be linear

MIN_GRADIENT_THRESHOLD Minimum gradient value for a point to be used

MAX_POINT_DISTANCE_Y Minimum y-distance for a new point to be used

MAX_POINT_DISTANCE_X Minimum x-distance for a new point to be used

THREAD_NUMBER Number of computing threads

V.A.4 Sequential Improvements

Having this basic concept of the algorithm implemented we needed to test it of course. But our first tests where not so successful, the car lost the line quite often. To prove our concept we replaced the Pi with an ordinary laptop, using a USB UART adapter. With this setup the algorithm turned out to be working really good, and the car stayed on the line every time, even with the driving motor running at full speed. The main difference between those two setups is the computation power, which is obviously way higher on the laptop than on the Raspberry Pi. After some performance analysis we figured out that the computation of a single frame takes around 4 ms on the laptop and 170 ms on the Raspberry Pi. We also figured out that our camera gives us 15 frames per second (FPS), so a new image is provided approximately every 66 ms. This means that we are only processing every third image on the Raspberry Pi and that is the reason for the poor performance. So there are two things to be done: first of all shorten the calculation time per image and second make sure every frame of the camera is processed and therefor really used, so we are no longer wasting information. Improving the calculation time also improves our reaction time, since thats exactly the time it takes to get an image of an event (e.g. an approaching turn) and to calculate what the car should do. So the faster the computation time is the faster the car can drive. Ensuring every frame gets calculated does not improve the reaction time, but it improves the overall throughput and therefor every 66 ms a new steering angle would be present. This also improves the feasible speed of the car.
To save computation time we disabled the software Gaussian Filter. It turned out that it can be easily replaced by changing the focus of the camera, so it produces a blurred image. That is exactly the same thing the Gaussian Filter would do, but in hardware rather than in software. It can be seen, that a sharp image without Gaussian Filter makes the algorithm very vulnerable to noise, but as soon as the focus of the camera is changed the algorithm recognizes the line without problems.
Another optimization is that the search range is restricted. Before the whole row of the image was searched for the highest gradient, now only the area of the X-Position found in the row below ± 80 Pixels is searched. The highest gradient is assumed to be on the line. This assumption is reasonable, since the line always is continuous and won't change so fast. This method only works if the pixel found in the lowest row of the image is on the line, and not noise. In reality this turned out to not be a problem since if there is

a correct line on the image it needs also to be in the lowest row of the image and it is identified correctly.
Furthermore only gradients grater than 25 are considered. This helps to recognize if the line has left the image to either the left or the right side of the image. In those cases it is possible that a row has no pixel on the line and therefor no gradient grater than 25. In this case the coresponding line gets neglected, saving further computation time due to less points on the line. Even more important is that this makes the algorithm less vulnerable to noise in the case that the line leaves the image to one of the sides. Without this any random noise above the point where the line leaves the image would be interpreted as the line and affect the steering calculations in a bad way.
With all these improvements we managed to bring the computation time down to approximately 100 to 110 ms per frame, which is pretty good, but still not good enough to capture all frames. But that approach only uses one core of the four cores the Raspberry Pi provides.

V.A.5 Exploit Parallelism

Since it is not possible to catch every frame of the camera with just one thread on the Raspberry Pi we need to exploit the parallelism of the Pi. So we first tried to make OpenCV use OpenMP to parallelize it's computations. That seemed to work, it brought the computation time per frame down to 80 ms. But that is still not faster than the 66 ms latency of the camera. So still further improvements need to be done. But since this implicit parallelism seems not to work well together with the explicit parallelism described in the next part we had to deactivate the OpenMP version again.
So we developed our own way to make use of the four cores the Raspberry Pi has. The idea is to use multiple threads that each grab a frame, process it and propagate the resulting steering angle to the UART output and the Nano board. With one computation taking 100 to 110 ms two threads are sufficient to calculate each image. The way this works is as follows:
The main thread creates two worker threads and sleeps then. The worker threads grab the next frame generated by the camera in a synchronous way, meaning one image is always processed by only one thread. Now the thread processes the image and calculates the steering angle. After that the thread performs a check whether the calculated frame is still in the same order than the sequential stream of images. If not, e.g. the calculation of this frame took to long and the next frame has already finished, it is neglected. Usually this is not the case, since the frames all approximately take the same amount of time to be computed. Than the worker thread wakes the main thread through a condition variable and propagates the calculated steering angle. While the worker waits for the next frame of the camera the main thread generates the UART output and sends the calculated steering angle to the Nano board. The main thread starts to sleep again, while the worker thread continues to process the next image.
With this technique the worker threads alternately grab the images provided from the camera, all frames are used and no information is wasted. Since the Raspberry Pi has four cores this method could also be run with four worker threads instead of just two. In our case, there is nothing to gain by using four threads rather than two, since the camera only produces 15 FPS. So for example with three threads one thread would always be waiting, so the algorithm degenerates

to the two thread version. But if a different camera with 30 FPS would be used it would be helpful to use four worker threads. This way even all the 30 FPS could all be calculated and used for the line detection. Figure 3 shows how all images of the camera can be calculated using two threads. Note that the time between two images and the time for calculating one image is on scale. For simplicity this shows only a part of the whole program and only the necessary communication. It can be seen that after each computation there is enough time for the thread to wait for the next image.

VI Conclusion

To conclude, the car followed a closed track without getting out of line in both directions. The maximum set speed was about 85% of the maximum possible speed. With a few modifications on the camera, e.g. a higher frame rate of 30 FPS, or a wider viewing angle, or a higher computation power on the Raspberry Pi, the full motor speed could had been reached.

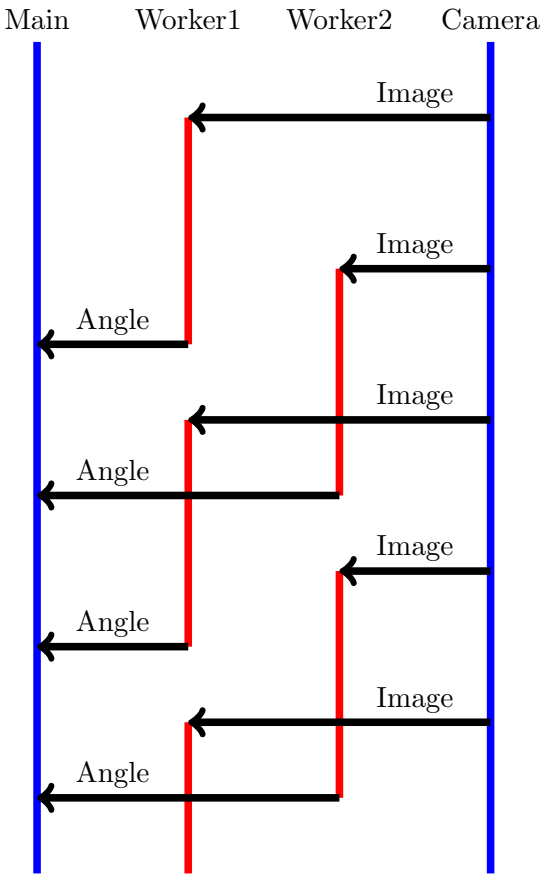

Figure 3: Multithreaded calculation