PROJECT REPORT

NTNU

TITLE:

Portal Runner

CANDIDATE(S):
OLE-MARTIN HANSTVEIT (460014)
OSCAR HERMAN KISE (460016)
MORTEN LERSTAD SOLLI (254801)

DATE:	COURSE CODE:	COURSE TITLE:	RESTRICTION:
11/30/2017	IE303812	Real Time Programming	None

STUDY PROGRAM:	PAGES/APPENDIX:	LIBRARY NO.:
Engineering - Automation (015AU)	51/10	None

SUPERVISOR(S):
Blindheim, Ivar
Strazdins, Girts

SUMMARY:

This is the project report for "Portal Runner", the final project in Real Time Programming. The purpose of this project is to make an autonomous car that can navigate and fetch objects using image recognition. Control of the vehicle should be operated from a GUI on an external client. This client should display images from a camera on the car.

This is solved through use of real-time and concurrent programming, 3D modelling and image processing. The program is built around the use of threads and the concept of thread-safety. The 3D modelling is used to design a gripping mechanism and brackets for various components. Image processing is used to detect colors and shapes.

The car manages to drive through portals of different color, chosen from the external client. There are no problems when detecting colors in either dark or light environments. However, it struggles to determine which portal is closer to the car. The gripping mechanism was not finalized and is therefore left out of the final product.

Thread-Safety was one of the main priorities and was maintained throughout the whole project.

TABLE OF CONTENTS

SUMMARY

This is the project report for "Portal Runner", the final project in Real Time Programming. The purpose of this project is to make an autonomous car that can navigate and fetch objects using image recognition. Control of the vehicle should be operated from a GUI on an external client. This client should display images from a camera on the car.

This is solved through use of real-time and concurrent programming, 3D modelling and image processing. The program is built around the use of threads and the concept of thread-safety. The 3D modelling is used to design a gripping mechanism and brackets for various components. Image processing is used to detect colors and shapes.

The car manages to drive through portals of different color, chosen from the external client. There are no problems when detecting colors in either dark or light environments. However, it struggles to determine which portal is closer to the car. The gripping mechanism was not finalized and is therefore left out of the final product.

Thread-Safety was one of the main priorities and was maintained throughout the whole project.

TERMINOLOGY

Concepts

Real-time	- Real time programs shall guarantee a response within a pre-specified time constraint
Concurrent	- Concurrency is when several tasks will run simultaneously, by running them in overlapping time periods.
HSV	- This is a color model which separates true color from light intensity. Hue is the color, Saturation is how dark it is, and Value is the lightness.
BGR	- This is the same as the RGB color model, but in a reversed order.
Stall current	- The maximum current drawn when the motor is applying its maximum torque.

Notation

GUI	- Graphical User Interface
BGR	- Blue, Green, Red
HSV	- Hue, Saturation, Value
I/O	- Input/output
PWM	- Pulse Width Modulation
SPI	- Serial Peripheral Interface
USB	- Universal Serial Bus
IDE	- Integrated Development Environment
V	- Voltage
A	- Ampere
DC	- Direct Current
HMP	- Heterogeneous Multi-Processing

EMMC	- Embedded Multimedia Card
UART	- Universal Asynchronous Receiver/Transmitter
GPIO	- General purpose input/output
IRQ	- Interrupt request
ADC	- Analog-to-digital-converter
I2S	- Inter-IC Sound
I2C	- Inter-Integrated Circuit
RAM	- Random access memory
GB	- Gigabyte
LPDDR3	- Low Power Double Data rate type three
HD	- High Definition
RPM	- Rounds per minute
Oz/in	- Ounce per inch
Kg/cm	- Kilogram per centimeter
IP	- Internet protocol
mAh	- milliampere hour
g	- gram
CAD	- Computer Aided Design
Ms	- millisecond
Uart	- Universal Asynchronous Receiver/Transmitter
kHz	- kilohertz
MIMO	- Multiple Input Multiple Output
WEP	- Wired Equivalent Privacy
WPA	- Wi-Fi Protected Access
CAD	- Computer Aided Design

1 INTRODUCTION

The objective of the following project is to get a better understanding about principles and methods of real time programming. The learning outcome of the course is to be able to develop real-time applications in object-oriented programming environments. This mandatory project is the basis for the evaluation of the course. (NTNU u.d.)

The project, Portal Runner is a car driven autonomously and manually controlled. The objective with the car is to maneuver through portals and pick up an object to be brought back to the starting position. A web camera is attached to the car to help the car to steer through the portals specified beforehand by color.

After successfully maneuvering through the set of portals, the car will pick up a specific object detected by the camera and an IR sensor. The object is gripped using a set of grippers handled by a single servo motor and lifted using a more powerful servo motor.

The setup for the car is done by selecting different settings in a graphical user interface (GUI). The settings include which color of the portal the car should drive through and what kind of object to collect, specified by shape and color.

Because this is a real-time project, a specification is set for the system. The system must be able to process at least 10 images per second. This means that the process of one image must be completed within 100ms.

This report will include a description over components used and explain the methods of the project. The chapter about the methods will explain more thorough about what the objective of the vehicle is, and how the different tasks are being solved. The results of the methods will be displayed, and discussions about accomplishments, troubleshooting, room for improvement, etc. will occur. The discussion will argue about the use of some methods over others and consider possibilities for improvements about the project.

2 THEORETICAL BASIS AND BACKGROUND

This chapter will document theoretical background about software, hardware and methods used in the project. The documentation presents what kind of program and language that were used for the object orientated programming, and necessary libraries used that was relevant for the project. The different principles and methods about real-time programming is explained.

2.1 *Java*

Java is a concurrent and object-oriented programming language that is widely used for creating computer programs and industrial applications. One of the big advantages with java is that code which is written in java and compiled can run on any computer running Windows, Linux, and MacOS.

2.2 NetBeans

NetBeans is an IDE for creating computers program and applications in different programing languages. It supports programing language such as Java, PHP, C/C++ and several others. NetBeans IDE is a cross-platform IDE, which means it can run on several Operating systems.

2.3 OpenCV

OpenCV is a library of programming functions mainly aimed at real-time computer vision. It is free for both academic and commercial use, and can be used by C++, C, Python and Java (OpenCV u.d.).

According to Nvidia, "OpenCV is the leading open source library for computer vision, image processing and machine learning, and now features GPU acceleration for real-time operation" (Nvidia u.d.).

Originally developed by Intel, it was later supported by Willow Garage and is now maintained by Itseez.

2.4 jSerialComm

jSerialComm is a communication library for Java. The library is made with the intention of a straightforward way to connect to and use the serial ports on a computer, independent of what operating system that is used. (Fazecast 2016)

2.5 Arduino

The Arduino is a microcontroller with different type of analog and digital outputs, there are also boards with ethernet ports and SD-card readers, and many different shield options. Arduino also has its own IDE named "Arduino IDE" and is made in java. The programing language for the Arduino is ArduinoC, which is the C programing language with extra libraries.

2.6 Odroid

Odroid is a small form factor computer which can run different distributions of Linux, but the standard OS is Open Android, which is Odroid's own Operating system. The basic I/O interfaces of the Odroid is USB-ports, ethernet, and HDMI.

2.7 PC-Schematic

PC-Schematic is a CAD software used to draw electrical wiring diagrams. The program has electrical symbols stored in the menu, making it easy to draw diagrams. The CAD generates automatically table of content, terminal connections and fills lists with product data used in the diagrams.

2.8 Concurrency

The principle behind concurrent programing is to run several tasks of a program at almost the same time. Each task will be assigned a thread, where the scheduler or executor decides which thread will run at any given time. If we have a single cored processor the executor will switch between threads either by time or events. By using concurrency in our program, can we can optimize the use of the CPU by fully utilizing the speed and cores of the processor. (González 2016) (Wellings 2004) Threads

2.9 Threads

Threads makes it possible to run several tasks at the same time by concurrently running them. A program or a process can consist of several threads completing different tasks. The threads need to be executed after they are created. This can be done either through an executor or a scheduler.
There are two ways to make a thread. We can either extend the thread class or implement the Runnable interface in the class. Each class that extends thread will be created as a thread, this means every object of the class will be its own thread.
For objects that shall run all the time it is smart to extend the Thread class, since then the object will always have a thread to use.
Contrary to Thread, when implementing Runnable we don't bind a thread to the object that is created. The object is given a thread each time it shall complete a task, as long as a thread is available. A hidden ThreadPool class is managing the usage of threads for the Runnable objects.
This makes it more efficient to implement the Runnable interface for tasks that won't run all the time. Because the runnable objects will share threads and use less recourses. Another advantage of implementing the runnable interface is that we can inherit form other classes. (Wellings 2004)

2.10 Thread Safety

Thread safety is an important concept for making our program work without faults, when making a concurrent application.

There are mainly four states we most try to avoid. A *data race* is when two or more objects try to write to the same variable at the same time. *Deadlock* is when several tasks awaits a common resource they wish to access, but another task is currently using the resource, making it unavailable. *Livelock* is when two tasks changes states due to the other. *Recourse starvation* is when no one get the resources they need. (González 2016)

Semaphore

Semaphores is way to make objects safe. A semaphore controls the access to a resource that is used by more than one task. A semaphore makes sure that only a specific number of tasks can access the resource at the same time. It hold permits and trough the *"aquire()"* and *"release()"* method, threads can gain and give up access. If the semaphore doesn't have any more permits, the asking thread will be denied access until another task releases its permit. If the semaphore only holds one permit, it acts as a mutual exulting lock.

Synchronized

The synchronization method can be used to either synchronize individual methods within an object, or synchronize the whole object itself. Synchronized is a mutual excluding method, which means only one task can access it at any given time, and everyone else trying to access it must wait. If a thread is writing to a "put method", all other threads trying to get the value or write to it themselves, most wait for turn. When synchronizing a mutating method, it is important to remember to synchronize its corresponding "get method" as well.

If we want a task to read a resource only after it is updated, we can use the *"wait()"* method. Then the resource can be updated by another task that will call the *"notifyAll()"* method after the update, telling the first thread it is ready to be read. When a thread is put in wait, it no longer holds the lock for the resource. This means another thread can mutate the object.

When a *"wait()"* is used, it is important for the other tasks to call the notifyAll() method after it is done. If it doesn't the program can end up in a deadlock state. (González 2016) (Wellings 2004).

2.11 ***ThreadPool***

The ThreadPool consists of a collection of all threads that are created when the application starts. It can be defined in the ThreadPool how many threads it should contained. It is important to keep in mind that the more threads in the ThreadPool, the more recourses are used. When executing a runnable task, it is given a thread from the pool to execute the task. When the task is done, the thread will return to the pool for another task to use. If there are no threads available the task will have to wait for another task to complete and return the thread. (González 2016)

2.12 ***Executor & Scheduler***

Executors are used for managing threads within an application. When using an executor, we can decide when certain tasks shall be executed. This is done by calling the objects "*run()*" method. When executing a task, the executor will acquire a thread from the ThreadPool and assign it to the Runnable task. If there are no available threads, the task will have to wait in a que until a thread is free. We can make a Runnable object and send it to the executor, which will manage the thread handling. Threads managed by the executor are thread-safe as long as the methods of the tasks are implemented correctly.

Tasks can be scheduled to run one time or periodical. To do this, we use a scheduler which can be part of the executor. When using a scheduler, we can say that a specific task shall run each 100ms, or run ones after 10 minutes and then never again. (González 2016)

2.13 Event

An event is used to signal from one class to another, when a desired state is reached. For example when new data is available to be read. The Event class usually uses predefined variable states, such as "UP" and "DOWN". The class has some methods that are used, such as; "set*()*" sets the state to UP, "reset*()*" to set the state to DOWN, "toggle()" will change the state, and "wait()".

The "wait()" method are used by tasks waiting for a specific state. When the other task calls set(), the first task will be awaken and execute its code before resetting the state. (Blindheim, Blackboard 2017)

2.14 Even Listener

EventListeners are used to trigger an event when an input is detected. This can be when we write on a keyboard, each keystroke triggers an event from an event listener. The listener then triggers an event which will start a task. When making an event listener is an important feature of the listener that it needs to react quickly, and if it needs to start a lengthy task, this should be done in another thread. (docs.oracle.com u.d.)

2.15 Serial Communication

Serial communication is a way of transferring data over one line. The data will be sent one bit at a time over the same line or bus. When using serial communications, are there two ways to define the transfer speed. Baud rate, which is how many times a second the line changes state. Bits per second, which is how many bits can be sent each second. The most widely used serial interface is the USB. USB supports full duplex communication, which means data can be sent and received at the same time. (taltech.com u.d.)

2.16 TCP

Transmission Control Protocol is a protocol for connection-oriented communication. Connection is established through a three-way handshake where the client sends a request for the server to open a socket. Then an acknowledge is sent back to the client which again responds with an acknowledge. This makes for failsafe communication between the client and the server. The TCP Header is 20 bytes long and consists of source and destination – port, sequence number, acknowledge number, etc. illustrated in Figure 1.

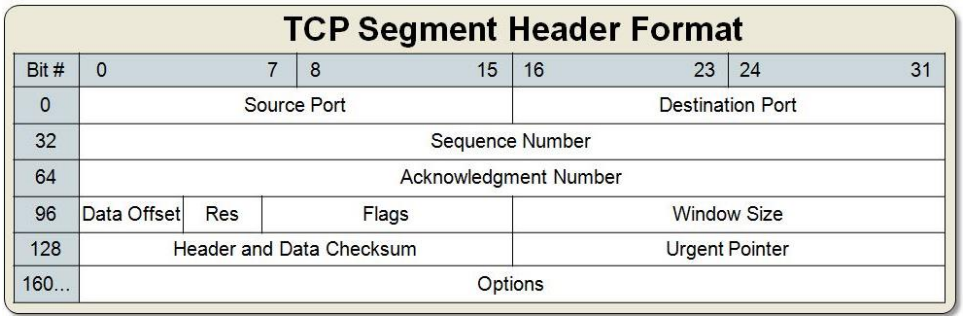


Figure 1: TCP Segment Header Format

TCP guarantees deliverance of all packages. If the package is lost along the way or corrupted, the sender will resend the message again. When a package is sent, the sender will wait for an acknowledgement. If no acknowledgement is returned, the sender will

eventually resend the package. When the package is received it is checked for errors. If it contains any errors, there will not be sent any acknowledgement message. Likewise if a package is lost, no acknowledgement will be returned.

TCP is used when it is critical that all messages are received, such as when loading a web page, or when we want to send a command for a program to do something. (James F. Kurose 2013)

2.17 UDP

User Datagram Protocol is a form of connectionless transport, this means that we don't need to establish communication before we start to send our message. When the program writes to the UDP socket, the socket will send a "DatagramPacket" containing the data and 8 bytes header containing; source port, destination port, length and a checksum, as illustrated below in Figure 2.

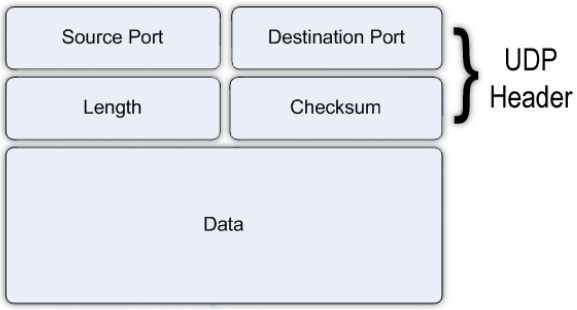


Figure 2: UDP Header Format

UDP Contains no deliverance guarantee and the sender will never know from the receiver if the package is received. If we encounter corrupt data in a package, the package will simply be discarded. The DatagramPacket can also be received in incorrect order. UDP is used when we want a high refreshment rate with as little latency as possible, and when it does not matter if we lose some of the packages. An example for this can be video streaming where we want the next image as fast as possible. (James F. Kurose 2013)

3 MATERIALS

The following chapter displays the materials intentionally meant for the project. The materials that were not used is marked as "not used". The information about the components will include technical specifications and other relevant information for the project.

3.1 *Sparkfun RedBoard R3*

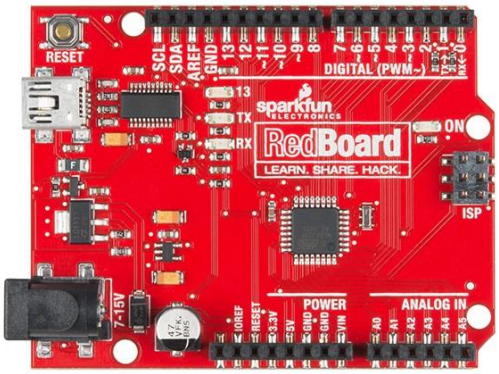


Figure 3: Sparkfun RedBoard R3 [1]

The RedBoard (Figure 3) is a custom board combined with unique features. It has the simplicity of the UNO's Optiboot bootloader, the stability of the FTDI and the shield compatibility of Arduino UNO R3. The boards hardware consists of 14 digital I/O pins, 6 of these pins are equipped with PWM, 6 analog inputs, one serial port, SPI and external interrupts. Programming of the RedBoard is executed by using the Arduino IDE with a USB Mini-B cable. It can be powered by this cable or through the barrel jack. The power regulator can handle voltages in the range from 7 to 15VDC. (Electronics u.d.)

3.2 *Arduino Motor Shield Rev3*

Figure 4: Arduino Motor Shield Rev3 [2]

The motor shield (Figure 4) is a dual full-bridge driver which is designed to drive inductive loads like relays, solenoids, DC and stepping motors. It has two motor drivers, channel A

and channel B, that can supply each own DC motor. Each of these drivers allows the user to control the speed and direction of the motor. The operating voltage is from 5 to 12V and by using external power supply can each channel supply 2A. (Arduino u.d.)

As the shield is stacked on an Arduino, are there some pins that are always in use. When addressed, can the direction of the motor change, adjustment for the speed can be specified (0-255), control of start/stop for the motors and the current flow on each channel can be tracked. The following pins are used regarding the channels A and B. (Sarafan u.d.)

Function	**Channel A**	**Channel B**
Direction	Digital 12	Digital 13
Speed (PWM)	Digital 3	Digital 11
Brake	Digital 9	Digital 8
Current Sensing	Analog 0	Analog 1

3.3 *Odroid XU4*

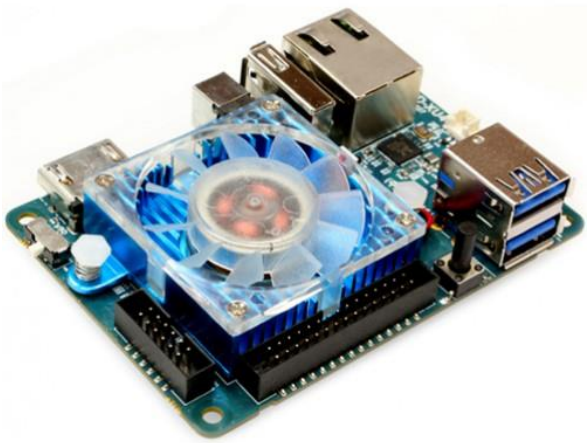


Figure 5: Odroid XU4 [3]

The Odroid XU4 (Figure 5) is a new generation of computing device. It is small, energy efficient and powerful. The board offers open source support, which allows it to run operating systems like Ubuntu 15.04 (Linux), 4.4 KikKat and 5.0 Lollipop (Android). The available I/O connections are 2 USB 3.0, 1 USB 2.0, PWM for the cooler fan, UART for serial console Ethernet RJ-45, 30Pin: GPIO/IRQ/ADC and 12Pin: GPIO/I2S/I2C. The input power of the device is 5V. Available RAM memory is 2 GB LPDDR3. (UK u.d.)

3.4 *Logitech Webcam C910*

Figure 6: Logitech Web camera C910 [4]

The web camera (Figure 6) offers a 1080 HD camera lens with autofocus. It is equipped with a microphone, activity light and a flexible clip/base. (Logitech u.d.)

3.5 *L-shaped Micro DC Gearmotor*

Figure 7: Micro DC Gear Motor [5]

The operating voltage of the motor (Figure 7) is from 3 to 7.5V DC. Without any load is the maximum speed 180RPM, the stall current is 2.8A and the stall torque is 11.11 oz/in. (RobotShop u.d.)

3.6 *Micro Servo Motor SG90 (not used)*

Figure 8: Micro Servo Motor SG90 [6]

The micro servo (Figure 8) has a range of angle rotation of about 180 degrees (90 in each direction). Its operating voltage is from 4.8V to 6V and the torque is 1.8 kg/cm. The hardware wire connections are ground, 5V+ and PWM. (ServoDatabase u.d.)

3.7 *Servo Motor MG995 (not used)*

Figure 9: Servo Motor MG995 [7]

Operating voltage of the servo (Figure 9) is from 4.8V to 6.6V and the stall torque is 9.4kg/cm (4.8V) and 11kg/cm (6.0V). The servos operating angle is from 0 to 60 degrees. The hardware wire connections are ground, 5V+ and PWM. (Pro u.d.)

3.8 *Maxpower Battery Pack*

Figure 10: MaxPower Battery Pack [8]

The battery pack (Figure 10) consists of six 1.2V battery connected serially, which makes the total output voltage 7.2V. The current of the pack is 4000mAh and the weight is 440g. (RCbutikken u.d.)

3.9 *Turnigy 5A SBEC*

Figure 11: Turnigy SBEC Voltage converter [9]

The Turnigy SBEC (Figure 11) is a DC-DC regulator which supplies constant 5V/5A and operates with voltages from 8 to 26V. The regulator is protected against reverse polarity on the input and the output voltage is interference-free. (Hobbyking u.d.)

3.10 Pololu Analog Distance Sensor (not used)

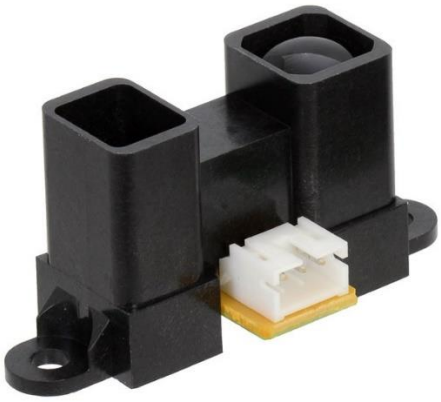


Figure 12: Pololu Analog Distance Sensor [10]

The sensor (Figure 12) measures distances from 4 to 30 cm and has an operating voltage from 4.5V to 5.5V. The resolution of the sensor increases the closer the sensor gets to the object and the signal output from the sensor is analog. The update period is 16.5ms ± 4ms. (Corporation u.d.)

3.11 RFID Reader (not used)

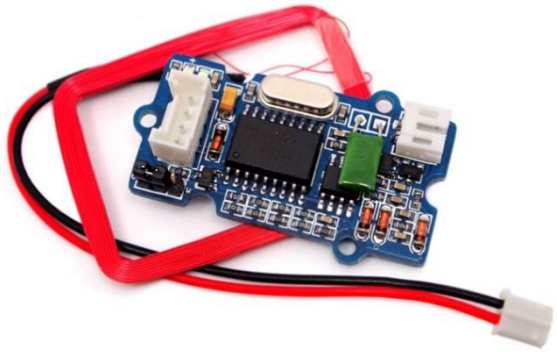


Figure 13: RFID Reader [11]

The RFID (Figure 13) reader operates at the frequency level 125 kHz and the operating voltage is from 4.75V to 5.25V. It has a reading sensitivity at maximum 7cm distance from the RFID tag. The available output formats are Uart or Wiegand. (Seeedstudio u.d.)

3.12 RFID tag (not used)

Figure 14 RFID tag [12]

The RFID tag (Figure 14) is operating at 125 kHz ± 6kHz in room temperature. The diameter of the disk is 20mm ± 0.5mm and the height is 0.6mm ± 0.2mm. The tag holds a unique 64-bit number to be read by a RFID reader and the communication protocol is EM4100. The communication protocol and frequency must match with the RFID reader. (RS Components u.d.)

3.13 802.11N WIFI adapter

Figure 15: Wifi adapter [13]

The wireless USB adapter (Figure 15) has a transfer rate up to 300Mbps and is compatible with the standards; *"IEEE 802.11n"*, *"IEEE 802.11g"* and *"IEEE 802.11b".* The device approves the use of MIMO technology and the internal intelligent antennas provides high transfer rates, high stability and a wide coverage zone. The adapter supports 64/128-bit WEP and WPA/WPA2 encryptions. (EDUP u.d.)

3.14 **Netgear N300 WiFi Router**

Figure 16: WiFi Router [17]

This wireless router (*Figure* 16) operates in the 2.4GHz band with a transfer speed of 300Mbps over wlan. It is compatible the standards; *"IEEE 802.11n"*, *"IEEE 802.11g"* and *"IEEE 802.11b".*

3.15 **Spray paint**

Figure 17: Spray paint from Biltema [14]

The spray paint (Figure 17) is an overall spray for outdoor and indoor use. (Biltema u.d.)

3.16 MP3 Module (not used)

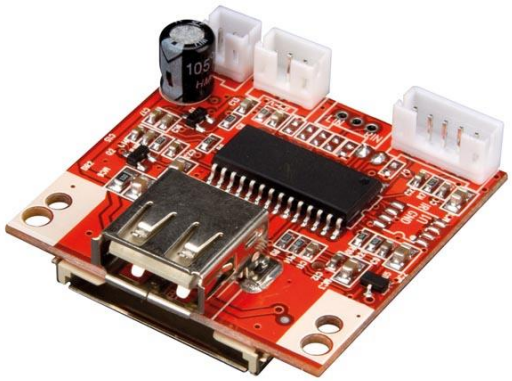


Figure 18: MP3 Module [15]

The MP3 module (Figure 18) can play mp3 files stored on a SD card or a USB stick. It supports 128kb/s, 192kb/s, 256kb/s and 320kb/s variable bit rate mp3 formats. The operating voltage is 5VDC or 9-12VDC. The module has several features like, pushbuttons for *"next/previous"* song and *"play/pause"* of the mp3 file, a LED indicator for when the song is on pause or play, continuous looping of all the tracks and the playback starts when the module is powered on. (Velleman u.d.)

4 METHOD

The following chapter will explain the plan for the project, how it is organized, procedures for testing and descriptions of the methods used to fulfill the intentions for the vehicle. The project plan diagram is attached under Appendix A, giving an overview of the main tasks and the expected time per task. Topics covered below concern the various 3D designs, construction of the portals, functioning of the code, serial communication and description of the hardware connections. Results of the methods are described in chapter 5, Results.

4.1 Project plan

The initial plan for the project is to drive a car, both manually and autonomously, through a predefined number of portals of assorted colors, and lastly to pick up an object of a specific shape and color. The setup for the cars objective is set in an external GUI. The car is maneuvering through these portals with image processing, by live video stream from a web camera in front of the car. Each portal is equipped with its own unique RFID tag, to be scanned by a RFID reader attached to the car. This allows to keep track of which portal the car has driven through, and keep track of the number of portals completed.

A speaker will play a sound when a portal has been passed, to indicate if it was correct or incorrect. When all portals are completed, will the speaker enlighten the user about how many portals of correct color the car managed to pass, and engage search for the object. The car is searching for a preset object by the camera, and an IR sensor placed in the front of the car facing down, will inform when the object is detected. The car is supplied with a gripper mechanism to grab the object, and lift it up from the ground. Lastly is the object brought back to the initial position for a drop off.

4.2 *3D design*

All 3D designs are done by using the CAD software, Autodesk Fusion 360.

4.2.1 Design of the gripper

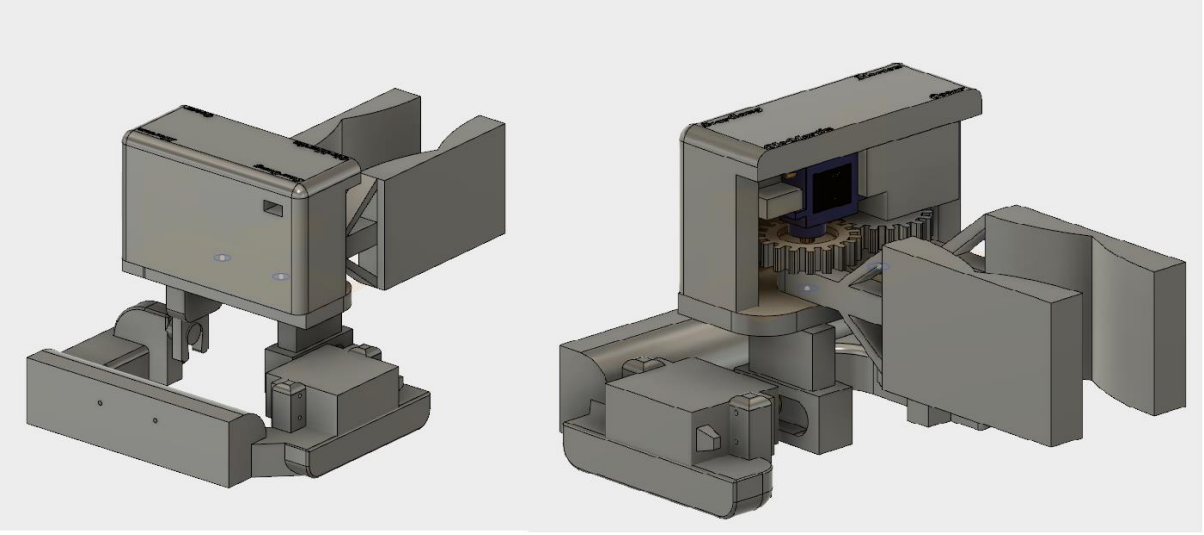


Figure 19: The final design of the gripper. Picture taken from Autodesk Fusion 360.

The gripper mechanism (Figure 19) is made to both grab and lift an object. By using gears to control the gripping mechanism, would only a single servo fulfill the gripping movement. The inside of the claws is attached with rubber, making the objects stick easier to the claws. Due to weight from the claws and the grabbed object, is a larger servo used for the lifting mechanism. The design can be separated in four part:

The lifting platform

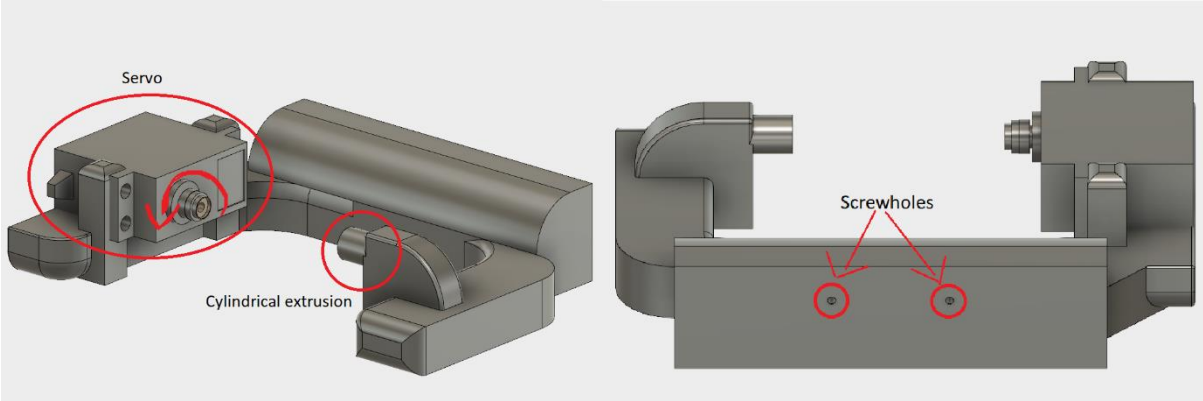


Figure 20: The lifting platform with explanations.

The lifting platform (Figure 20) has a MG995 servo mounted on its right side when looking forward from the cars perspective. The objective of the servo is to tilt the gripper platform (Figure 21) to give the ability to lift the objects grabbed. The platform has a cylindrical shaped extrusion on the opposite site from the servo, which allows the gripper platform to rotate smoothly around the same axis as the servo rotation. The lifting platform itself is mounted in the front of the car with two screws screwed into the two holes illustrated on Figure 20.

The gripper platform

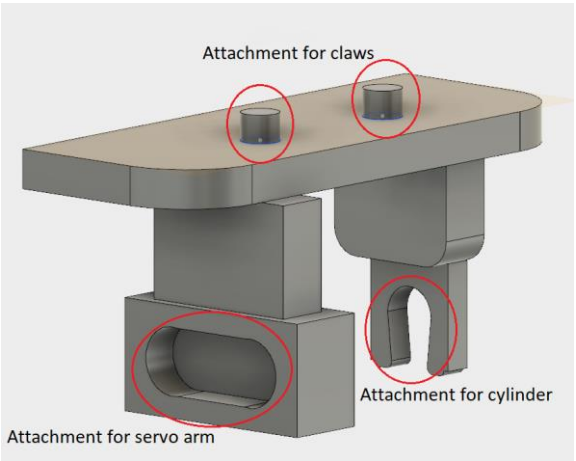


Figure 21: The gripper platform.

The gripper platform is attached to the lifting platform as illustrated on The plugs on top of this platform are cylindrical shaped, which lets the claws (Figure 22) rotate around their axis. The illustration indicates where the servo arm and cylinder from the lifting platform is placed.

The claws

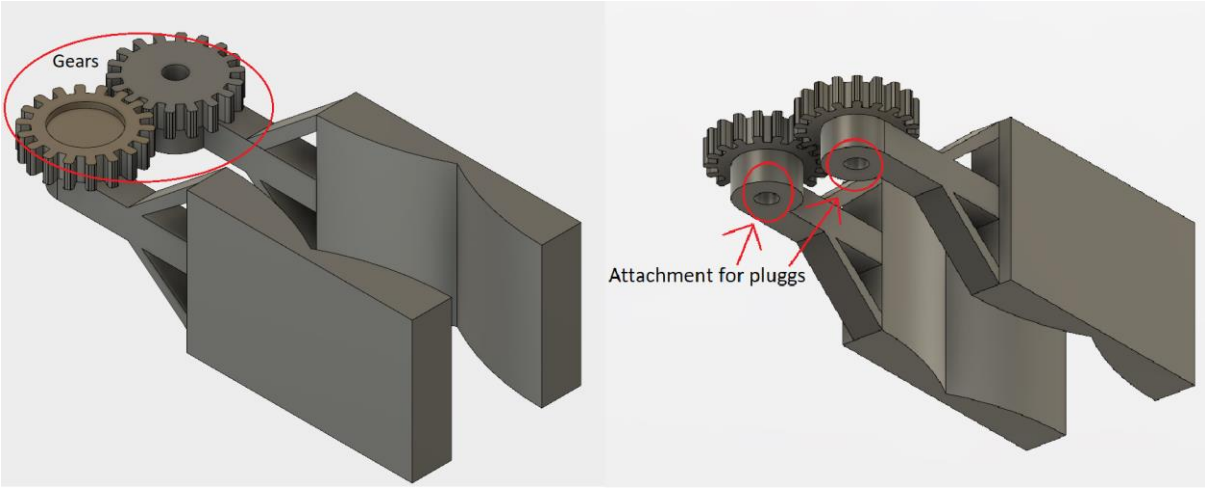


Figure 22: The claws controlled by the gears.

The movement of the claws (Figure 22) are done by a servo mounted with a flat disk on top of the right gear. The rotation of the servo allows the claws to grip and release an object. There is a hole under each gear, which are made for attachment for the plugs illustrated on the gripper platform (Figure 21).

The lid for the claws

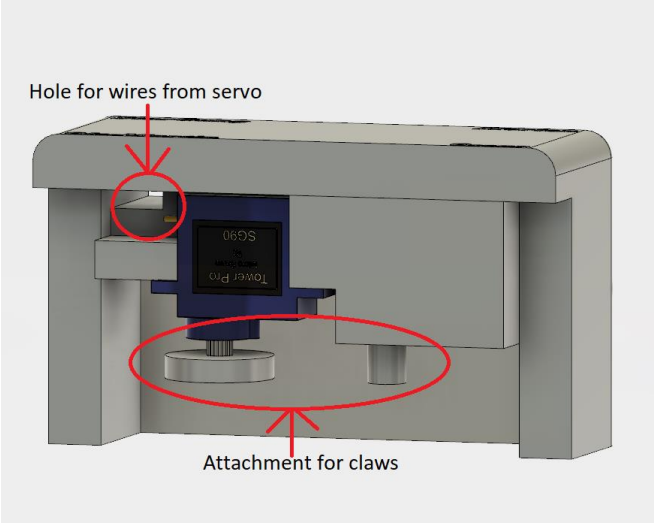


Figure 23: The lid for the claws.

The servo with the disk is mounted inside the lid (Figure 23) to be placed on top of the gripper platform (Figure 21). The lids objective is to keep the gears firmly placed on the plugs with the disk from the servo, and the plug extrusion inside the lid. The opening on the back of the lid is a passage for the wires from the servo to the connection point on the motor shield.

4.2.2 Bracket for web camera and IR sensor

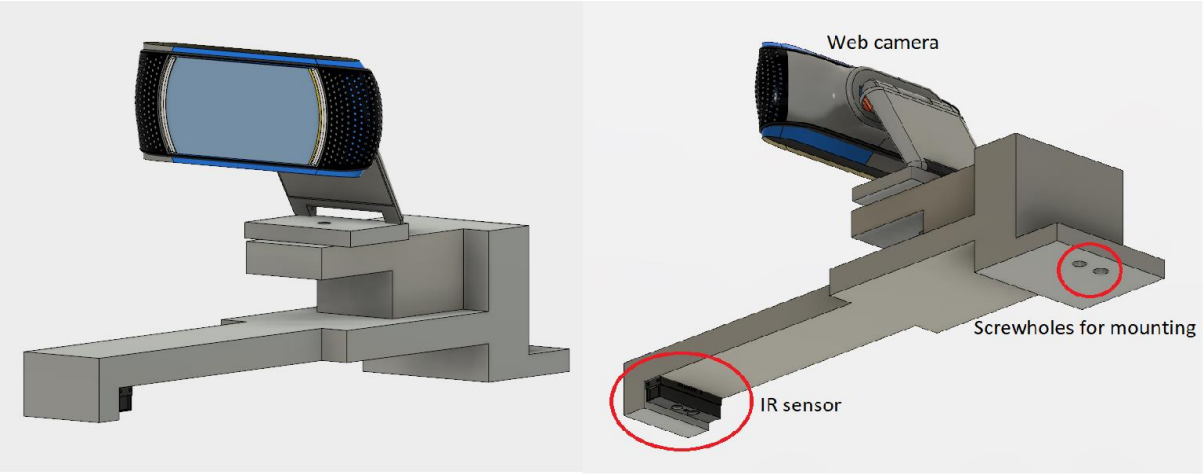


Figure 24: Bracket for the web camera and IR sensor. Picture taken from Autodesk Fusion 360.

The bracket (Figure 24) is designed for the camera to get a good overview of the track. The IR sensor attached facing down, measuring in between the claws, which makes it easy to detect an object. The whole bracket itself is attached to the metal platform on top of the car, mounted by two screws indicated on the illustration.

4.2.3 Battery holder

A bracket was designed for the batteries on the car. These are designed with holes to attach it to the car. There are also slots in place to be able to mount a lid on top of the batteries. See *Figure* 25 below for the design;

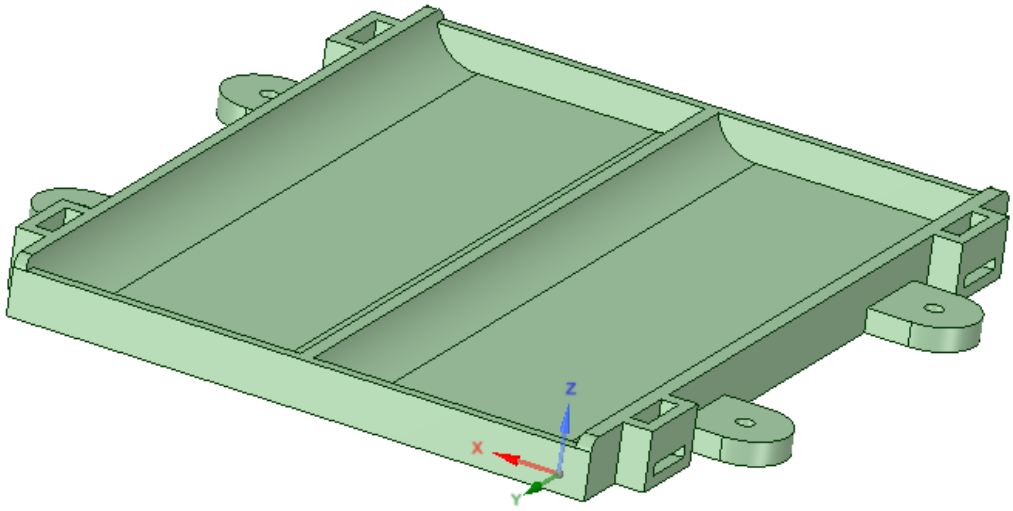


Figure 25: Bracket for the batteries on the car.

4.3 3D print

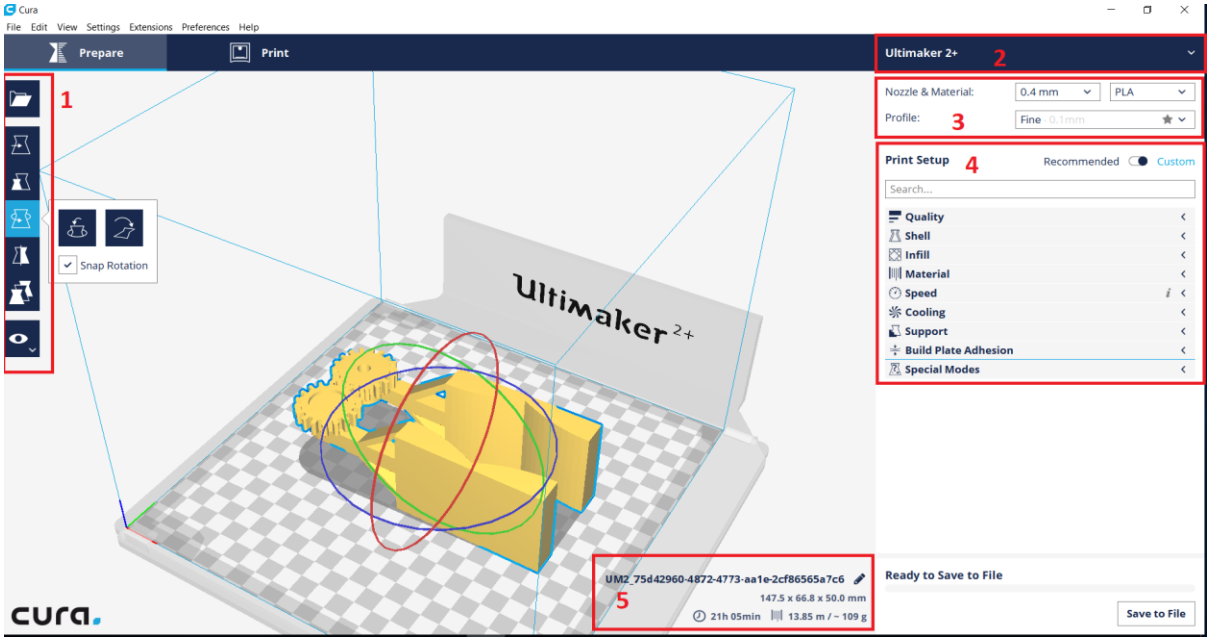


Figure 26: Setup for 3D-print of the claws in Cura 2.7.0

The designs were prepared for printing using the Cura 2.7.0 software (Figure 26). The panel on the left side (*marked as "1"*) displays the setting for the figure itself. These options let the user rotate, scale and move the object. On the top right is where the printer is selected *(marked as "2"*). Below that is where the settings for the printing material is selected (*marked as "3"*). The type of material and size is selected here and the thickness of the print can be defined under *"Profile"*. The print settings (*marked as "4"*) is preferred to be set to *"Recommended"* setup for non-experienced users. For the more experienced users can *"Custom"* setup be selected. The custom setup allows the user to select how much fill (in percent) the print shall have, where support is needed, speed, etc. The last rubric *(marked as "5"*) displays the calculated time for the print, dimensions and name of the file.

4.4 Portals

Figure 27: The portal for the car

The design of the portals (Figure 27) was made for the car to have enough space to pass through, with reasonable margins. The portals were made from wooden planks and spray painted. A colored paper was placed in the middle of the portal to be used for color detection, instead of detecting the color of the portal itself. This made it easier for the camera to detect the middle of the portal. The setup in the GUI will let the user select which portal to drive through based on the color.

4.5 Code

This subchapter will explain the initial code made for the project and how it will be implemented. There are parts of the code that are not used, but can be used for future extensions of the project. Class diagrams for the implemented code can be found in 51 D. A sequence diagram of the main application can also be found here. The main objectives for the code is explained below.

The car needed to be able to recognize colors and shapes to drive through portals of assorted colors and pick up objects of different shapes. This was solved by making code for reading and handling the image taken by the web. Methods were used for color detection, shape detection and pixel handling among others.

There was also a need for communication between the Arduino, Odroid and the external computer. The microcontroller needed to be able to receive inputs for driving and gripping objects, as well as sending sensor data back to the Odroid. Serial communication over USB was used for this, but I2C bus was considered as an alternative. Commands needed to be sent to the Odroid from an external client. Video streaming was also required from the Odroid to the client. To do this, TCP was chosen for sending commands, and UDP for sending video streams. It was decided to do this, because both protocols had desired features ideal for their purpose.

The external client's objective was to present the operator with an image, showing what the car sees, as well as sending commands back. Here it was made a GUI with buttons for start and stop, options for color detection, and a picture frame at the bottom showing the video stream.

To make the car move, an Arduino was used as an I/O device. The function of this microcontroller was to execute the commands given from the Java application on the Odroid, and return the relevant sensor data.

The necessary concurrent and Real-time approaches should also be a part of the processes discussed above. Making the program concurrent by implementing threads and tasks where it is needed, and making sure the process is thread safe by using semaphore and synchronizations correctly. One measure was to use events when it was needed to signal other threads of a change of state. All this had to be done without making the program too fine or coarse -grained.

Below you can see in more detail how each part of the code and the classes are made to function, and how they are connected.

4.5.1 Image Processing

One of the main goals for the car is to make it autonomous. Automotive navigation of the vehicle is accomplished by using image processing on raw video from a webcam attached to the car. The image processing is implemented in Java by using the OpenCV library. In this project, version 3.1 of OpenCV was used. To achieve a Real-time functional system, a requirement of 10 processed images per second was specified (see introduction). This is fast enough for the car to navigate based on live input from the camera.

Image Processor

A Runnable thread named *ImageProcessor* is in control of processing an image. It runs at a specified number of loops per second. It is set to 30 runs per second to utilize the 30 FPS capacity of the camera. This thread doesn't have any methods, it distributes tasks to several other classes. This thread is run by a *ScheduledExecutorService* in the main method. This ensures that the submission of the Runnable task for *ImageProcessor* is thread-safe. There is no further need to ensure thread-safety in this Thread because the tasks are implemented thread-safe in the classes used for each method.

For capturing images, the *WebCamHandler* class is used. After fetching the most recent image, the *ImageProcessor* will scan the image for a specified range of colors by using the *ColorDetection class. At this point, the ImageProcessor* will have a black-white image only showing white where the specified colors are found. Using this image, it will scan left and right half for white pixels. These values are used to navigate towards the center of objects by using the *MovementHandler* class.

In addition to this behaviour, a class named *ShapeDetection* was made. This was made to detect the shape of objects that it would pick up. However due to time restrictions, the feature was removed.

Video Capturing

A Thread called *WebCamHandler* captures video continuously. This thread initializes the camera by using the OpenCV library. The image resolution is set to 320x240, this resolution is good enough to detect important details. It is also a low enough resolution to run on Odroid without performance issues. Initially, 640x480 resolution was used, this proved to be too demanding for the Odroid. By testing the difference, a full loop of the "*run()"* method in *ImageProcessor* was reduced from 60-80ms down to 2-15ms. A test was run with resolution set to 320x240 with 1000 loops of the "*run()"* method, which gave an average loop-time of 5ms.

To use the video that is captured, the *WebCamHandler* thread contains methods for getting a new image from the camera;

```
// Reads current image and return it as a Mat
// Synchronized to prevent another method call that potentially could
// overwrite the Mat during another operation
////////////////////////////////////////////////////////////////////////////////
public synchronized Mat getMat()
{
    Mat frame = new Mat();
    if (camera.read(frame)){
        return frame;
    }
    return null;
}
////////////////////////////////////////////////////////////////////////////////
```

Figure 28: Synchronized method for retrieving an image from the camera as a Mat.

As seen on Figure 28, "synchronized" is used to implement thread-safety. This method is used to get a new image from the video capturing. It is used both for the *ImageProcessing* and for sending video to the external client by a thread named *VideoStreamServer*. Using "synchronized" here ensures that the method cannot be executed by both *ImageProcessing* and *VideoStreamServer* at the same time.

Color Detection

The class *ColorDetection* contains a method that processes a Mat of an image by searching for specified color values. A Mat is a class that represents a complex array that among other things, can store data of an image. First, the image is converted from BGR color model to the HSV model. HSV is different color model that handles darkness better by using Hue for color while Saturation and Value specifies the light intensity. This means that if the brightness in the image changes, the image Hue values remain the same.

After converting the image to HSV, it searches through the image for HSV values within the range specified. The HSV search parameters can be changed while active by a user from an external client. To find ideal HSV values for each color, calculators designed for this purpose could be used, such as the one made by a website called RapidTables (RapidTables u.d.). However, these were not ideal as the values were not accurate when tested with the camera. Therefore, a temporary GUI was made to identify correct values.

This GUI contains sliders for adjusting hue, saturation and value. It was used for finding the color range for each color used in the project. This GUI can be seen in where the sliders are located at the top of the picture.

When detecting colors, there will be small blobs of pixels scattered around. This is noise, which can easily be reduced. There are methods in the OpenCV library named; "*erode()*" and "*dilate()*" that deals with this issue. As explained by the website *Openframeworks.cc*, erosion removes a layer of pixels from every blob in the scene, while dilation adds a layer (Openframeworks u.d.).

Figure 29 illustrates the effects of using the "erode()" method.

Figure 29: Original image (left), after one pass of erosion (right). [16]

Old images can end up hogging all the computer memory if they are not handled correctly. Therefore, all images are released once they are no longer needed.

Movement Handling

To drive the car, the *ImageProcessor* uses a class called *MovementHandler*. This class is designed to handle the navigation based on number of pixels located in the left and right half of an image. When the input is processed and it determines the correct direction to move, it will command the Arduino to use the motors for the wheels. This command is

sent by using the Runnable *SendSerial.* The command consists of three words, first it indicates direction, followed by the speed for the left and right-side motors.

Shape Detection

To be able to pick up objects specified by shape, there needs to be a method for detecting shapes. This is done by the class *ShapeDetection*. It processes a thresholded image that contains a black background and objects as white pixels. First, it applies blurring to the image. This makes it more ideal for finding contours. Second, it finds the contours of each object. With these contours, it can determine the number of corners for each object. With the number of corners, it can identify the shape. In addition, it will check the width and height of objects with 4 corners to determine if it's a square or rectangle. Each detected shape is marked with a description, as well as a box around it. Due to restricted image quality, there is an offset for number of corners for each shape, as well as width to height ratio for square objects.

This feature is not included in the final code due to time restrictions. The shape detection feature is fully working, but some parts of the object fetching was not finished. The feature for shape detection is shown in Figure 30, by detecting objects on a sheet of paper;

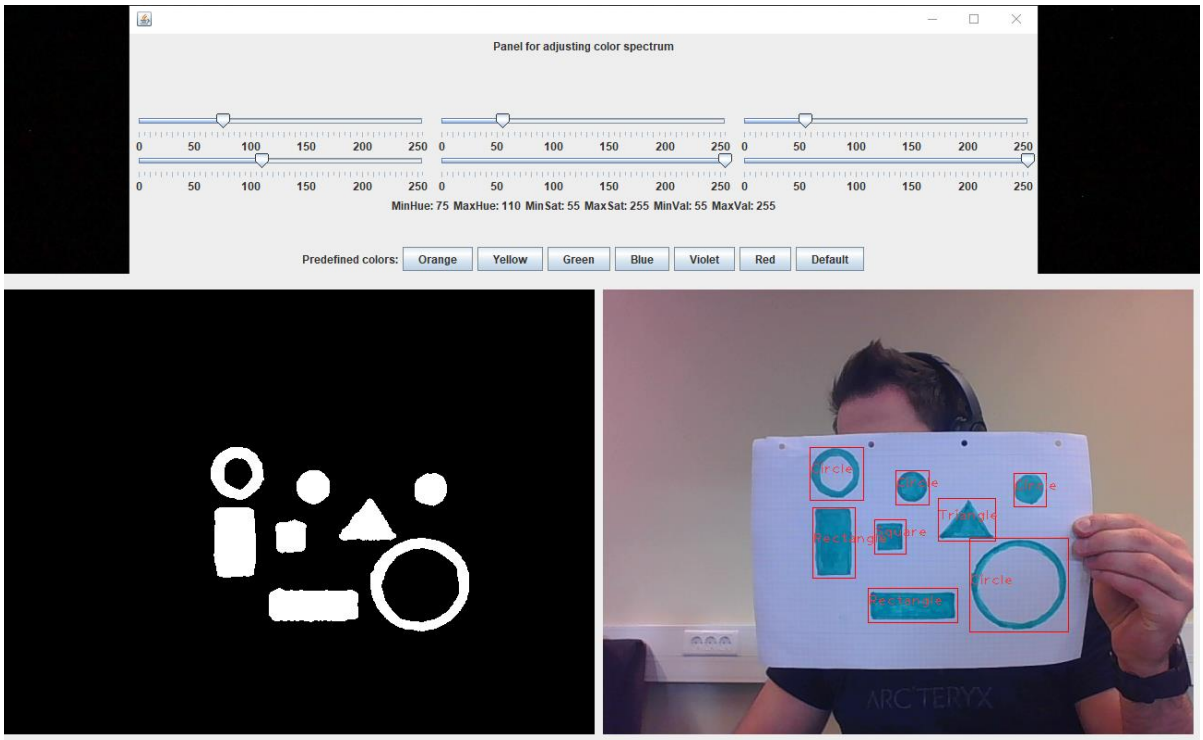


Figure 30: Picture from live video. Settings panel on top, detected objects to the left and shapes detected to the right.

4.5.2 Arduino Communication

The communication between the Arduino and Odroid is done through serial communication over and USB line. USB supports full duplex communicating which means we can send messages both ways at the same time. Messages are sent both to and from the Arduino. The Arduino receives commands from the Odroid telling it what to do, and the Odroid

receives sensor data back. The message consists of a string containing three parameters separated by a forward slash "/". The first parameter is a word telling what the next parameters are, the following parameters are integer values.

When sending to the Arduino the first parameter tells which switch case to go in to, and the next are variables such as motor speed. When sending from the Arduino to the Odroid, the first parameter tells which sensor the data in the next parameters comes from.

Serial communication is implemented in the Java program with use of the jSerialComm library for Java. (Fazecast 2016)

Through the *communication* class a connection with the Arduino is established. This class is initiated by the main class of the application when the application starts. It finds every open com-port on the system and checks it up against the port that is desired for communication. If it is a match, it will try to connect to this serial port.

To make the car drive, turn, stop, and grip there was needed to send commands to the Arduino. *SendSerial* is the class responsible for doing this. Through the use of the "*SendStringCommand()"* method, a string is build up by the three parameters discussed earlier. The string is then written to the output stream by a *printWriter.* The design of this class is thread safe. There is no need to synchronize the class nor its methods because it is only used by one task.

Sensor data for distance and RFID is needed from the Arduino. This is solved with three classes, *InputHandler, ReceiveSerial,* and *StringSeperator*.

The *InputHandler* delegates tasks regarding incoming data, separating string messages into parameters, and send the parameters to the correct handler. Data is retrieved from the listener when an event for received data is triggered. The data string is then retrieved from the listener and sent to separation. When the data is returned as substrings, the substrings will be passed along to the correct handler.

ReceiveSerial listens for inputs on the connected serial port. It creates an input stream and uses a "Data Listener Event" (see chap. 4.5.4) for continuous listen for incoming data. When incoming data is received, it will trigger an event that notifies the *InputHandler* that data is ready to be extracted. *ReceiveSerial* is not a thread, but it utilizes the data event listener which makes the class continuously listen on the serial port.

When the *InputHandler* sends strings to the *StringSeperator*, the string will be separated in to several substrings. The *StringSeperator* checks the input character by character to build the substrings. It will start on a new substring each time a forward slash is detected. When it is done it will return an Array containing the substrings to the *InputHandler*. The *StringSeperator* is a versatile class that can be used in other parts of the code as well.

4.5.3 Odroid to external communication

The Java to Java communication is done through a Wi-Fi connection between the Odroid and the external GUI. Two different protocols are used for video streaming and sending data. The User Datagram Protocol is used for video streaming and the Transport Control Protocol is used for data. The reason for choosing UDP is that a fast connection is needed, and it is not essential if some frames are lost, as long as they are up to date.

The UDP server is Implemented on the Odroid for sending a video stream to the external GUI. It implements the Runnable interface, which makes it a thread that will be executed 30 times a second. This makes it a timed task for getting a smooth and stable video transfer between the two computers.

The *VideoStreamServer's* function is to stream video in real time to the external user interface. A DatagramSocket is opened by the UDP server, and DatagramPackets containing a picture is sent from this socket. Pictures are acquired from the *WebCamHandler*. When a picture is acquired, it is converted to a bufferedImage and wrapped in a DatagramPacket for sending. The sent picture has a resolution of 320x240 pixels.

There is a UDP client on the external user interface. The client implements the Runnable interface, which makes it a thread. It runs continuously receiving pictures from the Odroid. Pictures are received as bufferedImage and are converted to a Mat object for resizing from 320x240p to 640x420p. The resized image is then converted back to a bufferedImage, ready to be used in the GUI.
Resizing the image makes it larger, but also a bit blurry. This is done to use as little resources as possible to send the image on the Odroid, and because a sharp image is not that important in the GUI.

For sending commands such as start, stop, and which colors the car should detect, a TCP connection is used between the Odroid and the external user interface. The TCP server is located on the Odroid. This means the external GUI must make a connection to the car before it can start to send.

The server on the Odroid named *ClientListener* is a thread that is continuously listening for inputs from the GUI. When data from the GUI is detected by the listener, it will forward the message on to the *ServerDispatcher.*
The dispatcher is a thread that lies in wait until the listener calls for the method "*dispatchMessage()"*, which wakes the thread and forwards the message. The dispatcher will then retrieve the message, make a String out of it and pass it along to the TCP input handler. Then the *ServerDispatcher* will go into wait again until a new message is received. When the TCP *InputHandler* gets the message, it then sends it to an instance of the *StringSeperator* to get out each substring of the message. The substring is then return, and the input handler then handles the commands, either telling the car to start or stop, or setting the colors it should drive after.

The TCP client is located on the external client. This external client is responsible for establishing connection towards the TCP server and send the commands. Sending commands is done through a printWriter. This is a thread-safe operation because there is only one thread writing to it.

4.5.4 Event

Events are often used to signal a change of state. If we want a task to run when there is a change of state in another task, it is smart to use an Event. In the project, there is an event class. In example, it is used to signal a change of state in the *ReceiveSerial* class. When incoming data is detected, *ReciveSerial sets a state to "UP". As can be seen in* Figure 31*.*

```java
/**
 * Reads the inputstream and creats it to a string, when the messages is
 * read the inputEvent will be set to UP.
 */
public void readInputStream() {

    boolean dataRecived = false;

    while (data.hasNext() && !dataRecived) {
        dataString = data.nextLine();
        System.out.println(dataString);
        dataRecived = true;
    }
    inputEvent.set();
```

Figure 31: ReceiveSerial, showing set event when data is received

The *InputHandler* (which is waiting) will awaken, then execute code before resetting the state to "DOWN". Which we can see in below.

```java
inputRecivedEvent.await(Event.EventState.UP);       // Wait until input is recived
inputRecivedEvent.reset();                          // Resets event to DOWN
dataString = input.getDataString();
```

Figure 32: From InputHandler, showing await and reset.

The *Event* class has methods for setting an event to "UP", reset the event to "DOWN", or to toggle between the two states. It got a method named "*await()",* which will put a thread into a waiting state. Then the "*notifyAll()*" will awaken the sleeping threads.
It is a god method for securing thread safety where one or more tasks are dependent on each other, and can be used everywhere a change of state should trigger something.

DataEventListener

The "*eventListener()"* is a method in the jSerialComm library. It listens on the serial port for incoming data. When incoming data is detected it calls for the "read*InputStream()".* The method makes it possible for the class to continuously listen to the serial port for incoming massages. The implementation is illustrated on below.

```java
/**
 * This methode listens for inputs continously, when an input is detected an
 * lisinig event is triggered.
 */
public void eventListener() {

    port.addDataListener(new SerialPortDataListener(){

        @Override
        public int getListeningEvents() {
            return port.LISTENING_EVENT_DATA_AVAILABLE;
        }

        @Override
        public void serialEvent(com.fazecast.jSerialComm.SerialPortEvent event) {
            if (event.getEventType() != port.LISTENING_EVENT_DATA_AVAILABLE) {
                return;
            }

            readInputStream();

        }
    });
}
```

Figure 33: ReceiveSerial, *DataListenerEvent implementation.*

4.5.5 Handling sensor data

Handling sensor data received from the Arduino is the *SensorHandler's* responsibility. It is similar to the StorageBox object which we have had about in class.
The function of the *SensorHandler* is to store the sensor variables from the RFID sensor and IR sensor in sub classes.
It contains synchronized methods to make sure that the threads reading from and writing to the object can't interact with it at the same time, making it a mutual excluding object. This is done to not get corrupt data when reading from the object. It also contains an event to notify the reader when new RFID data is received.
This handler is meant to get sensor data from the *InputHandler* and the *MovementHandler* is supposed to retrieve it.

4.5.6 Arduino

The Arduino is used as an input/output device. Setting motor speed and direction, servo position, and reading sensor data. To do this, the Arduino also needs to send and receive data with the Java program.

Incoming data strings are received from the USB port and separated into substrings. This is done similar to what is done in the java program. A string of 3 parameters is received, separated by a forward slash. The first parameter is a word which is mapped to a switch case, the two next parameters are variables which will be used inside the case. Functions for writing to outputs are placed inside the switch case. When a case is selected it takes in the variables and run the methods specified in that case.

The Arduino reads data from both the distance sensor and RFID scanner. Then a string will be built from the sensor data. The name of the sensor and the value is added as two parameters. It is then sent to the java program over the same serial line as commands are received.

4.5.7 External client

To control the car, an external GUI application was designed. This GUI communicates with the Odroid by sending commands to it, and receives a continuous video stream from it. The communication is established through a Wi-Fi connection to a dedicated router for this project. In the GUI, a user can specify the color of the portals the car should drive through. The GUI allow three portals to be specified, but the implementation is currently limited to the first color selected. The color selection of the second and third portal is not registered by the Odroid.

There is also an option for color and shape of the target object that the car should fetch. These options are not registered by the Odroid due to the feature not being included in the final code. It is included to suit future implementations.

There is an option for starting and stopping the car. Pressing the start button will update the selected parameters in the GUI and send these to the Odroid. The stop button sends a "stop" keyword that is handled by the Odroid to make the car stop.

Additionally, there is a toggle option for enabling/disabling both TCP and UDP connection. However, there is currently a bug where re-enabling the TCP will not be possible. This is due to the TCP-socket only being closed on the client side, not on the Odroid itself, when TCP is set to disabled. To fix this, the server-side (Odroid) must close the connection as well when it is closed from client-side.

It is also possible to toggle between manual and automatic mode in the GUI, this currently has no functions. It is included in the final code to suit future implementations.

The video stream is displayed in the GUI as a 640x480 resolution image that is continuously updated as new images arrive. The image is received in 320x240 resolution and resized to 640x480 to get the desired size.

Below is a picture taken of the GUI during operation;

Thread implementation

The incoming stream of images is received by a Runnable thread called *InputVideoStream*. This thread listens for incoming packets through a UDP connection. It expects a packet that contains an image as a BufferedImage. When it receives an image, it is converted to a Mat, resized to appropriate format, then turned into a BufferedImage again. The resized image is stored as a variable for the GUI to fetch when needed. This thread is considered thread-safe because there is a try-catch statement preventing the thread from becoming stuck when there are no incoming packets. The method for getting the resized image has no need for being synchronized due to only being accessed from one thread.

For a thread-safe initialization of the GUI, the initialization is scheduled for the event-dispatching thread (Oracle u.d.). This is done with the "*invokeLater()"* method, which will initialize the GUI when the thread is available. This is the thread which also handles GUI events such as actions to do when a button is pressed.

```
//Schedule a job for the event-dispatching thread:
//creating and showing this application's GUI.
javax.swing.SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        new GUI(videoStream, socketClient).createAndShowGUI();
    }
});
```

Figure 34: Code for scheduling the initialization of the GUI

It was added an *ActionListener* to the buttons in the GUI to handle the actions bound to the buttons (Figure 34). The methods that are run by the listeners are designed to be fast to prevent stopping the entire event-dispatching thread. If this was not taken into consideration, the entire GUI could potentially freeze while the thread was stuck in an operation.

To update the image in the GUI, a *Timer* is used to schedule an update every 33ms. This time is specified to achieve 30 frames per second which is enough for a smooth image.

4.6 Wi-Fi setup

Between the Odroid and the external GUI, there is a Wi-Fi network for data transfer. The network consists of one Wireless router, an USB Wi-Fi adapter for the Odroid, and a laptop with wireless network.

The router is configured as a DHCP server which hands out IP-addresses to the Odroid and the laptop. The easiest way to connect the Odroid is to write the following command into the terminal window: nmcli d wifi <WiFiSSID> password <WiFiPassword> iface <WiFiInterface>.

4.7 Hardware connections

The hardware connections consist of four parts, a RedBoard, two Arduino motor shields and an Odroid XU4. The Redboard has two motor shields stacked on it, making the stack three levels high. The wiring diagrams are found in Appendix C.

The first level is the RedBoard, which is communicating serially via a USB cable, with the Odroid.

The second level of the stack is the first motor shield. The motor drivers provide power for the two front motors of the car, connected to the output channel A and B.

The third level is the second motor shield, controlling the two rear motors of the car, by the output channels A and B. The 8V battery is connected to the power input, providing all levels of the stack with power. The two servos used on the gripper mechanism are attached on the output slots 5 and 6. These slots consists of three pins, +5V, ground and PWM. The signal from the analog distance sensor used to localize the object to be gripped, is placed in the input slot 2. This slot consists of three pins, +5V, ground and analog input.

The Odroid receives input power from the regulator, which is connected to the battery. All three of the available USB ports is connected to a device. The connections by USB is a web camera, an WIFI adapter that provides the Odroid with wireless internet and the RedBoard communication cable.

5 RESULTS

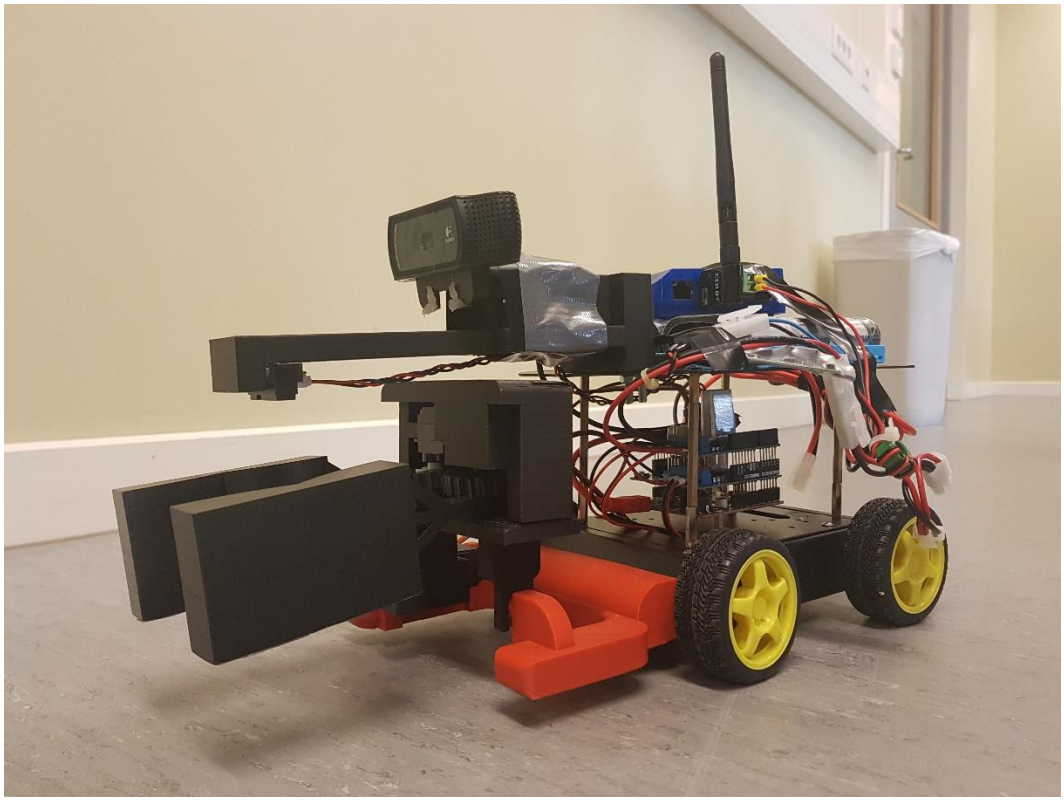


Figure 35: The finished product of the car

The chapter will present the performance of the methods in chapter 4 and the result of the finalized hardware.

Figure 35 displays the final product of the "Portal Runner" car. The main objectives of project had to be reduced due to lack of time. The vehicle is manually started from the GUI. The car will now drive autonomously through portals of a specified color, until another command is given. This means that when a portal is passed, it will start a new search for the next portal immediately. The functionality of picking up objects was dropped. Even though much of the code and hardware was finished for this part, it would take too long to implement and finish the rest of the car.

5.1 Code

5.1.1 Communication

The communication to the Arduino over USB works nicely. There is a limitation of how fast the Arduino can receive messages because of its architecture. The Arduino also got a timeout on 5ms, which means no messages can arrive with less time than this in between.

Connecting to the serial port and sending from the Odroid works without problem. Sending strings and separating them in the other end was a good method to establish communication between the two.

Not all classes in the serial communication was used. This was due to project was decreased in size, so we no longer needed the sensor data from the Arduino. The classes that are not part of the final code are the *ReceiveSerial, Inputhandler,* and also the

SensorHanlder. Even though the *StringSeperator* was a part of the serial communication it is being used in the TCP communication.

For communicating with the external client, sending video using UDP seems to be a good solution. As long as we have a stable Wi-Fi connection and sends about 30 frames per second, we get a smooth and consistent picture. To have the UDP server as a timed task makes sure that the Odroid sends all pictures it is supposed with a consistent time in between.

When testing the UDP connection it shows that when loosing Wi-Fi, the car will continue as before, without video. When the connection returns the video stream, it will reestablish itself after about 30-40 seconds. This is the same case for the TCP connection, where the client will try to send until the connection is back. This means that the "Portal Runner" car can operate on an unstable Wi-Fi connection.

The TCP connection is stable and secures that all commands are received by the Odroid. Sending string messages with several parameters was used here as well and works perfectly. This made for the *StringSeperator* to be used for the TCP communication. Making the *clientListener* a continuously listening thread that notifies the *InputHandler* when messages recives works as intended. We also manage to maintain trade safety within the TCP framework.

5.1.2 Image Processing

The implementation of image processing was successful. As mentioned in 4.5.5, the camera continuously captures images through a camera placed on the front of the car. This image is processed to detect colors that can either be pre-defined or defined by an external GUI. After detecting the colors, the Odroid can navigate towards the center of detected colors. This is done by sending commands to an Arduino which specifies parameters for each motor, which in turn makes the car move. Shape detection was also successfully implemented, but was not used in the final code. It was not used because the feature for lifting, which it was needed for, was not fully finished. As mentioned in 4.5.5, a temporary GUI was made to identify the best HSV values for finding each color. This worked exactly as desired, and resulted in exceptional quality of the color recognition.

To achieve a Real-time functional system, a requirement was set to process at least 10 images per second. This requirement was mentioned in 4.5.1 and is important for the car to react fast enough. The final system processes 30 images each second which is faster than required.

There are limitations for the program that were verified by tests. When the portal is rotated sideways toward the car, the car will drive straight into its leg. Another limitation is that it cannot decide which portal is closest, such feature is not implemented. Tests also checked the quality of color recognition in both dark and bright areas. This can be seen in the demonstration video, see 51 E and F. As mentioned in 4.5.1 about the *WebCamHandler*, a test for loop-time to the *ImageProcessor's* method "*run()"* was also performed. This test showed a substantial improvement of performance when the image resolution was reduced. A demonstration of the image processing can be seen in 4.5.1, Figure 30.

5.1.3 SensorHandler

The *SensorHandler* was never implemented in the final code and therefore not optimized towards the *MovmentHandler* and *InputHandler.*

5.1.4 Arduino

The Arduino have code for reading sensor data and sending it as it was supposed to. However, this code blocks are not part of the running loop because as the project was redefined it was no longer needed. The parts separating strings, the switch cases, and methods within the switch cases works perfectly. Taking in variables as motor speed instead of having fixed speed made the program more flexible and easier to use.

5.1.5 External client

To control the car, an external GUI application was successfully implemented. This client is fully operational for the current features it is designed to control. As mentioned in 4.5.7, interactive components for some features implemented in the future is included. These components currently have no effect besides visual. The GUI is shown in *Figure* 36 below.

Figure 36: External client application, finalized GUI.

Messages sent from the client are received by the server and correctly handled regarding functionality. The car responds to the commands in real-time with minimal delay. A router was dedicated for optimal communication speed.

There was a problem with closing TCP-connections server-side. This problem was mentioned in 4.5.7 and it prevented reconnecting to the Odroid when communication was closed. The problem was not resolved; however, the problem was identified. To fix it, there

could be a consistent verification of connection between server and client to identify a closed connection.

The image in the GUI is updated at a fixed rate of 33ms, which results in 30 frames per second.

5.2 Hardware

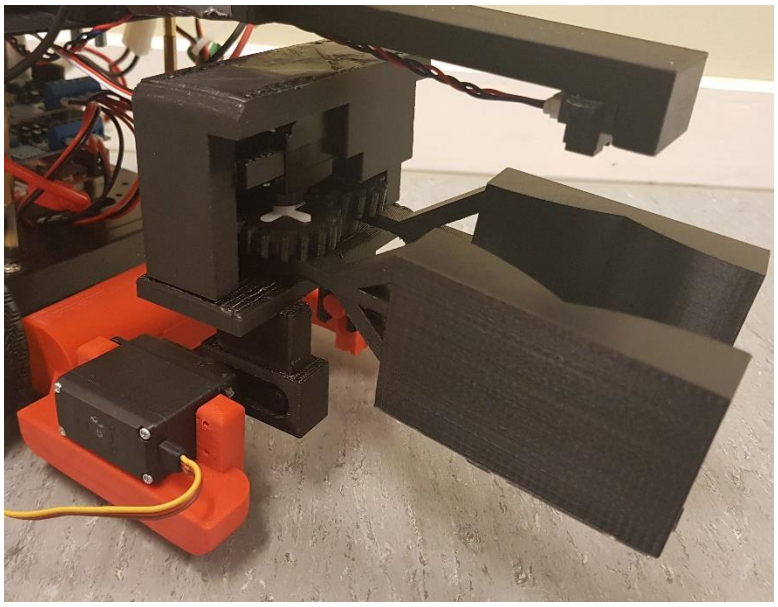


Figure 37: The gripper

This subchapter describes the quality and result of the gripper, camera bracket and the portals. An illustration of the gripper is shown in *Figure* 37.

5.2.1 Gripper

The lifting platform was mounted to the front of the car and the MG995 servo made a good fit in its slot on the right front extrusion.

The gripper platform fitted well on to the lifting platform and allows the platform to rotate around the axis of the servo without any complications. The claws became solid, but a little too big, leading the arms to tilt downward in the front section. The placement of the plugs made a good fit for the gears, but the diameter of the plugs became a little too small.

The claws were almost made identical, but mirrored. So, for the gears to fit and rotate, one of the arms had to be moved slightly in front of the other to make the gears function correctly. This lead to a minor problem, which caused the tip of the claws to not meet when brought together. This was fixed in the next version of the claws, leading the tips to meet accurately (Figure 38). The attachment for the servo to the gear was also changed from a cross to a disc, because a disc was easier to attach.

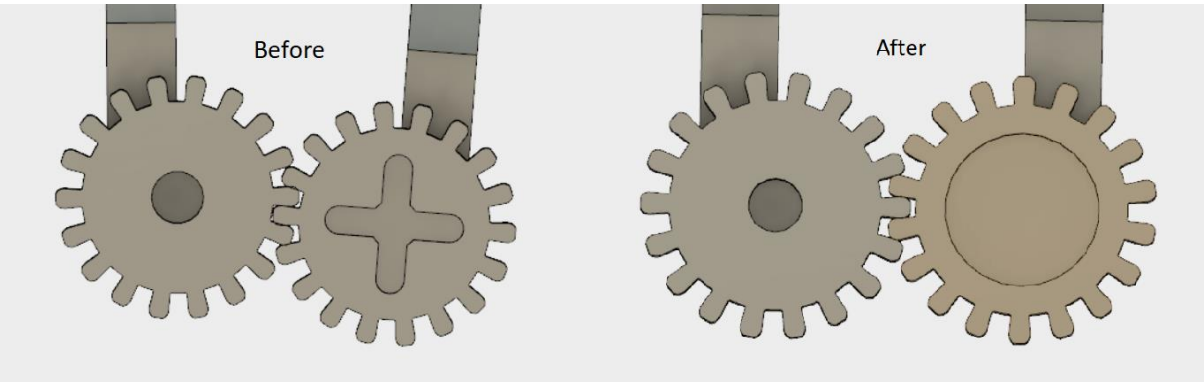


Figure 38: Left side shows the first draft and right side shows the final draft.

Lastly, the lid with the SG90 servo had the objective to hold the gears in place, but also manage to operate the gripping function. By this, the servo and the plug from the lid had to fit correctly in the slots on top of the gears. This was done successful, but a new problem appeared. The torque of the servo motor was not powerful enough to rotate properly all the time, causing the gears to struggle with the rotation. But, even though it managed to function, the grip strength was not strong enough to hold on to any type of object. So, after discussion with the group and the supervisor, became the decision to shelf this part of the project, due to lack of time until deadline.

5.2.2 Camera and IR sensor bracket

The bracket became a good fit for both the camera and the IR distance sensor. The distance sensor placement was right in the middle, between the claws, making it easy to detect any object. The placement of the camera was ideal for detecting the portals, but detecting the object became a conflict, due to lack of visual toward the ground by the claws. Though the gripper functionality was shelved for now, was there no necessary adjustments made to the bracket.

5.2.3 Portals

The portals were initially supposed to be 3D printed, and spray painted in assorted colors. This idea was scrapped in the preliminary stages, due to limited access to the 3D printers and the amount of time it took to print one portal. So, the portals ended up being made of wooden planks and use a color paper to attach to the front of the portal.

6 DISCUSSION

The following chapter will discuss the methods in chapter 4, and the results described in chapter 5. The discussion will cover the technical results achieved and the outcome of the project.

6.1 Code

6.1.1 Arduino Communication

The Communication to Arduino was made using serial communication. I2C was considered as an option in the beginning because the USB port on the Arduino Romeo (chap. 6.3.3) was broken. But the choice fell on serial communication when we switched to another Arduino and since the group was more familiar with this protocol.

It was then planned to use the RXTX library for serial communication in java. This was changed in advantage for the jSerialComm library which builds on RXTX, but is simpler to use and easy to find documentation and help for.

When the serial communication with the Arduino was established, there occurred a problem as the Arduino didn't always read the incoming data strings. This was due to the Odroid sending messages faster than the Arduino could read. The solution to this was to set a timeout of 5ms on the Arduino, this means that messages must have at least 5ms in between them. This works fine since the microcontroller receives messages with a frequency of 30Hz.

It could be considered to implement a method for checking if the port is still open before sending. To make sure the port always stays open and we don't have to restart the program if the connection is lost. This would be an advantage if the USB connector fell out it wouldn't be necessary to restart the whole program, but just reconnect the USB.

Everything having to do with inputs from the Arduino was cut short. This could be implemented in a later version of the project. The plan was to use this for counting how many ports was passed and measuring distance to an objected that would be picked up. The *SensorHandler* was supposed to take care of that data and notify users when it was updated. Much of the code is already finished, but not in use. Putting this into use with the framework for serial inputs could be a good way to continue development of this project and get new features.

6.1.2 UDP & TCP Communication

Both UDP and TCP is used for communicating with the external GUI.

We decided to use two different protocols for video streaming and sending commands. We could have used just one, but we believed the cost of using two different protocols were worth it.

The UDP have the advantage of sending messages fast and being connectionless, it also doesn't care if all packages is received. This could be a problem if we send command trough this protocol. It would be problematic if the stop was sent and the package was lost, resulting in the car not stopping. However, it is optimal for sending video streams where we only care about getting the latest packages and not care to retrieve the lost ones.

Contrary to the UDP, the TCP is a connection oriented protocol which encores delivery of packages. For sending commands this is desired, because if we lose a packet on the way it will be resent. For video, this will become a disadvantage which can make our video stream slower, and present us with old images. Especially if we have manual control video over TCP will become a big problem.

6.1.3 Image Processing

The implementation of image processing started off good. The concept was divided into small parts which allowed for testing each section apart from each other. The intention was to identify colors and shapes, and create a behavior for the car depending on these identifications. The implementation of these features was successful, although several problems were identified along the way. These problems are discussed further down in 6.3.4 about troubleshooting.

A requirement of how many images that needed to be processed per second was specified in 26, and the result was presented in 37. The requirement was to process 10 images per second, while 30 processed images per second was achieved. This resulted in a fast reaction time of the system, which led to good navigation in Real-time.

6.1.4 External client

The application for an external GUI was simple to implement due to lose coupling in the Odroid application design. The most demanding parts were to place GUI components correctly, and to update the image smoothly.

6.1.5 Arduino

On the Arduino, we receive variables to set the motor speed and server positions with. Another way to solve this could have been by having different methods for driving with fixed speeds. Doing this would have made it easier to send messages to the Arduino since we would only need one command, and we wouldn't need to separate a string.

But on the other hand, it would have been more work to tune the driving, having to re program the Arduino each time a change had to be made. Also sending variables makes the program more versatile and it uses less memory on the microcontroller.

Another change that could be done in the future, is to make use of the sensors and send sensor data back to the Arduino.

6.2 *Hardware*

The design and concept of the gripper was accurate, but the torque of the SG90 servo was not strong enough to function as intended. This caused the gripper to not be able the grip the object. Therefor would a stronger servo, like the MG995 servo be an improvement that would complete the gripper, making it fully functional. This process was started, making an improvement of the claws and adjusting the gripper platform (see chapter 5.2.1). By the time the adjustments for the lid were being processed, were there little remaining time until deadline. The conclusion was that the project would profit more from completing the real-time solutions and focus on writing a satisfying report, rather than completing the gripper and deliver a deficient report.

If the gripper would have been fully functional, the camera bracket would have had to be changed, making the camera able the get a view of the object to be picked up. The objectives of the car were probably a little too ambitious, when it came to the functionalities of the car.

The portals were first spray painted, but this caused issues with the image processing, causing the difficulties with detecting the middle of the portal. Therefor was a piece of paper placed in the middle of the portal, making the color easier to detect. This was a success, which lead to a better detection of the portals.

6.3 *Troubleshooting*

There were some issues during the project that are described below.

6.3.1 Motors

There were some difficulties in the beginning, about the motors that were time-consuming. It was experienced that the motors did not operate at the same speed. And the chip on the motor shield became very hot. There was mounted a heatsink in hope of eliminating this problem. The motor shield was changed, in hope of fixing the problem, but in the end, was it the motor itself that was defect. So, in total was two out of four motors changed, and this solved all the problems, making all four motors operate correctly. This also caused the chip on the motor shield to overheat, because the defect motors consumed to much current.

6.3.2 3D printing

Due to few available 3D-printers, it was rather difficult to be able to print, causing a lot of downtime before all the parts to the gripper could be put together. There were also many incidents where the print became dysfunctional, due to fault with printer.

6.3.3 Motor drivers

The car was equipped with Romeo V2 "All in one" Controller in the beginning. But there were problems with one of the motor drivers, which did not deliver enough voltage to the motors on this channel. There was no identical controller available, only one with a defect serial port, so the controller was switched to a Sparkfun RedBoard with Arduino Motor Shield stacked upon it.

6.3.4 Image processing

Several problems were encountered in the process of making image identification and behavior. These problems will be reflected upon in the paragraphs below.

There were issues installing OpenCV v3.3 on Odroid. After several attempts, a teacher was consulted for assistance. The teacher found a solution for installing OpenCV v3.1, which was then used. Because the names for methods and such being equal in v3.1 and v3.3, the code was compatible for both versions.

The code for image processing was first tested on PC and verified to be working. However, unexpected problems were encountered when running the same code on Odroid. The performance of the Odroid was much weaker than a laptop, which resulted in an overloaded system. To fix this, the resolution was reduced from 640x480 to 320x240. This reduced the workload enough to achieve a quick and stable system.

When switching over to Odroid for testing, another performance issue was also found. Around 40-50 seconds after the application was initialized, it crashed. It could not be re-initialized without restarting the Odroid. The issue was that the memory on the Odroid got filled up to the point where the camera could not capture images anymore. To fix this, all

old images that was no longer used were released. By releasing these, old data would no longer fill the memory, and the application ran smooth.

Another unexpected problem was discovered when running a real test of the system on the car. When testing the navigation, it seemed to be working properly. But sometimes the car would navigate onto one of the legs of the portal instead of the center. This problem was resolved. What happened was that when one of the portal legs would be out of the picture, most of the detected color would be at the other portal leg. It was solved by changing the color of the legs. This way, only the top bar would be found, which resulted in more correct identification of object center.

The car would also lose track of the object when it got too close. This happened because the camera had a limited vertical view. When the car got too close to the portal, the top bar of the portal would be outside the view. To solve this, a sheet of paper with correct color was placed at the center of each portal. This resulted in color detection until the car were halfway through the portal.

To make sure that the car would drive completely through the portal, the code was modified. It was designed to know when the portal straight in front of it by reaching a certain number of pixels found. When this was registered, the car would move straight forward for 1.5 seconds while pausing any other movement handling.

When passing through a portal, an error would occur if the next portal was positioned left of the camera view. When it started searching for the portal, it would turn right until found. This caused the car to see the previous portal instead, and drive back. Therefore, an improvement should be made to the object searching method. Instead of turning right until a new object was found, it could move left and right within a specified angle. This would prevent the car from turning 180 degrees and accidentally go back to the previous portal.

6.3.5 External client

There were not many issues regarding the external client application. The main issue was to properly close a TCP-connection as mentioned in chapter 4.5.7. Due to limited time before an approaching deadline, this bug was not prioritized to fix. It was not prioritized because it did not reduce functionality for a final presentation of the project.

7 CONCLUSION

The initial objective for this project was to make an autonomous car that would maneuver through portals of different colors, and fetch objects of various shapes. The most important focus was to maintain a thread-safe system which runs concurrently. We were too ambitious with our project plan, which later caused us to remove certain features of the car. Our time schedule (Appendix B) was altered during the project development.

As a final result, we made the car run through portals of different colors. We can decide which color to detect from a GUI on an external client, which can be changed while the system is running. Our specified requirements (see introduction) for system response-time was achieved with the final product. The car has no issue navigating toward the center of a portal, however it can't compensate for approaching it at a steep angle, as described in 5.1.2.

The feature for fetching objects was excluded in the final state of the project. Physical parts of the gripper were finished, and the corresponding software was mostly completed. The issue with this function was the torque of the servo, which controls the gripping movement. This caused difficulties with picking up objects, and due to the remaining time until deadline, other tasks were prioritized.

Serial communication between the Arduino and Odroid works within our requirements, regarding the system response time. As for using UDP for video streaming and TCP for sending commands, we are pleased with the result this gives us. We can maintain a concurrent and thread safe program, while getting a rapidly updated picture and secured transfer of the commands.

We are satisfied with using the Arduino as an input and output device. It is easy to stack the motor shields on the Arduino, which makes it better for controlling motors and servos. Receiving variables instead of having fixed speed gave us flexible control of the car, and we can easily implement a regulator in the future.

According to previous students, detection for some colors was hard to accomplish. In our experience, this was not an issue. We designed a GUI for finding ideal color values. This worked out well, and we did not experience any problems with this solution.

If there would be any future development of the project, then we suggest that the gripping mechanism should be finished. The search algorithm for finding portals could also be improved.

The outcome of this project gave us better understanding of real-time programming, importance of making a thread-safe system and how to find a balance between a fine and coarse -grained application. In addition to the relevant topics for the course, the group achieved knowledge about 3D design and image recognition. Personally, the group had an overall good experience from the project and gained new knowledge.

Work distribution:

Morten Lerstad Solli: Creating code for serial communication between the Odroid and the Arduino, for both directions. Creating code for UDP video streaming between Odroid and external client. Creating the communication on the Arduino and sewing together the Arduino program. Setup the Wi-Fi connection on the Odroid and the router.

Ole-Martin Hanstveit: 3D design of the battery bracket. Development of image processing (color/shape detection) and car movement handling based on images. Designed the external client including integration of the UDP video stream and implemented TCP communication between Odroid and the external client. Also integrated the separate Java codes to work as one application on the Odroid and implemented threads for the sections to run concurrently. Ran performance tests of the system.

Oscar Herman Kise: 3D design and print of the gripper and camera bracket. Constructing the portals. Creating the Arduino code for running the motors. Setup of the Odroid with necessary software. Created wiring diagram for all hardware connections.

8 REFERENCES

Arduino. *Arduino Store.* n.d. https://store.arduino.cc/arduino-motor-shield-rev3 (accessed 11 20, 2017).

Biltema. n.d. http://www.biltema.no/no/Bilpleie/Maling-og-lakk/Hobbylakk-2000023797/ (accessed 11 21, 2017).

Blindheim, Ivar. *Blackboard.* 2017. https://ntnu.blackboard.com/bbcswebdav/pid-156979-dt-content-rid-1242247_1/courses/194_IE303812_1_2017_H_1/IE303812_Thread%20Communication_Buffers_Atomics_Events_Broadcast_Blackboard_2017%281%29.pdf?target=blank (accessed November 26, 2017).

—. *ntnu.Blackboard.com.* n.d. https://ntnu.blackboard.com/bbcswebdav/pid-156959-dt-content-rid-1242025_1/courses/194_IE303812_1_2017_H_1/IE303812_Semaphores_2017.pdf (accessed November 2017).

Corporation, Pololu. *pololu.com.* n.d. https://www.pololu.com/product/2464 (accessed 11 20, 2017).

docs.oracle.com. n.d. https://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html (accessed November 27, 2017).

EDUP. n.d. http://edupwireless.com/product-1-1-4-high-definition-usb-adapter-en/137312 (accessed 11 21, 2017).

Electronics, Sparkfun. *Sparkfun.* n.d. https://www.sparkfun.com/products/13975 (accessed 11 20, 2017).

Fazecast. *Github.com/Fazecast/jSerialComm.* 12 5, 2016. https://github.com/Fazecast/jSerialComm (accessed November 2017).

González, Javier Fernández. *Mastering Concurrency Programming with Java 8.* Birmingham: Packt Publishing Ltd., 2016.

Hobbyking. n.d. https://hobbyking.com/en_us/turnigy-5a-8-26v-sbec-for-lipo.html?___store=en_us (accessed 11 21, 2017).

James F. Kurose, Keith W. Ross. *Computer Networking.* Harlow, Essex: Pearson Education, 2013.

Logitech. *Logitech.* n.d. https://www.logitech.com/assets/31650/2/c910gettingstartedwithguide.pdf (accessed 11 20, 2017).

NTNU. *NTNU.* n.d. https://www.ntnu.edu/studies/courses/IE303812#tab=omEmnet (accessed 11 20, 2017).

Nvidia. *Developer Nvidia.* n.d. https://developer.nvidia.com/opencv (accessed November 22, 2017).

OpenCV. *OpenCV.* n.d. https://opencv.org/ (accessed November 22, 2017).

Openframeworks. *openframeworks.cc.* n.d. http://openframeworks.cc/ofBook/chapters/image_processing_computer_vision.html (accessed November 26, 2017).

Oracle. *Oracle Java Documentation.* n.d. https://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html (accessed November 27, 2017).

Pro, Torq Pro & Tower. *Towerpro.* n.d. http://www.towerpro.com.tw/product/mg995/ (accessed 11 20, 2017).

RapidTables. *RapidTables.* n.d. http://www.rapidtables.com/convert/color/rgb-to-hsv.htm (accessed November 28, 2017).

RCbutikken. *RCbutikken.* n.d. https://rcbutikken.no/produkter/ni-mh-batteri/maxpower-ni-mh-72v-4000mah-deans-plugg.aspx (accessed 11 20, 2017).

RobotShop. *RobotShop.* n.d. http://www.robotshop.com/en/dfrobot-6v-180-rpm-micro-dc-geared-motor-with-back-shaft.html (accessed 11 20, 2017).

RS Components. n.d. http://uk.rs-online.com/web/p/radio-frequency-identification-rfid/8430800/ (accessed 11 21, 2017).

Sarafan, Randy. *Instructables.* n.d. http://www.instructables.com/id/Arduino-Motor-Shield-Tutorial/ (accessed 11 24, 2017).
Seeedstudio. n.d. https://www.seeedstudio.com/Grove-125KHz-RFID-Reader-p-1008.html (accessed 11 21, 2017).
ServoDatabase. *ServoDatabase.* n.d. https://servodatabase.com/servo/towerpro/sg90 (accessed 11 20, 2017).
taltech.com. n.d. http://www.taltech.com/datacollection/articles/serial_intro (accessed November 27, 2017).
UK, ODroid. *ODroid UK.* n.d. https://www.odroid.co.uk/hardkernel-odroid-xu4/odroid-xu4 (accessed 11 20, 2017).
Velleman. n.d. https://www.velleman.eu/products/view/?id=417492 (accessed 11 21, 2017).
Wellings, Andy. *Concurrent and Real-Time Programming in Java.* West Sussex: Jhon Wiley & Sons, Ltd, 2004.

9 PICTURE REFERENCES

[1] https://cdn.sparkfun.com/assets/parts/1/1/7/2/2/13975-04.jpg

[2] https://store-cdn.arduino.cc/uni/catalog/product/cache/1/image/520x330/604a3538c15e081937dbfbd20aa60aad/A/0/A000079_featured_2.jpg

[3] https://www.odroid.co.uk/image/cache/catalog/liymo/odroid/XU4/XU4-500x500.jpg

[4] https://secure.logitech.com/assets/31671/c910-front-view.jpg

[5]http://www.robotshop.com/media/catalog/product/cache/7/image/900x900/9df78eab33525d08d6e5fb8d27136e95/d/f/dfrobot-6v-180-rpm-micro-dc-geared-motor-with-back-shaft-1.jpg

[6] http://www.etechpk.net/wp-content/uploads/2016/08/micro-servo-motor-tower-pro-9g-sg90-com-acessorios-20181-MLB20185070463_102014-F.jpg

[7]https://img3.banggood.com/thumb/view/oaupload/banggood/images/CE/15/234b8cc5-f3b4-47dc-aa0f-070ff14e3f56.jpg

[8] https://rcbutikken.no/UserFiles/Products/4545_maxpower-ni-mh-72v-4000mah-deans-plugg_22.08.2014010904.jpg

[9] http://www.hellasdigital.gr/images/detailed/12/61sy7FwJs7L._SY355_.jpg

[10] https://a.pololu-files.com/picture/0J5788.1200.jpg?e76bc7fce22d2bbb4667acc943debf0e

[11] https://statics3.seeedstudio.com/product/gr125k_01.jpg

[12] http://media.rs-online.com/t_large/F8430800-01.jpg

[13]https://www.bazaargadgets.com/image/cache/catalog//products/computernetworking/networking/EDUPEP-MS8512300MbpsHDTVIEEE80211ngbWifiNetworkAdapter-SKU123596-1-800x800.jpg

[14] http://images.biltema.com/PAXToImageService.svc/byfilename/xlarge/36-418_xl_1.jpg

[15] https://www.velleman.eu/images/products/0/vm202n_detail.jpg

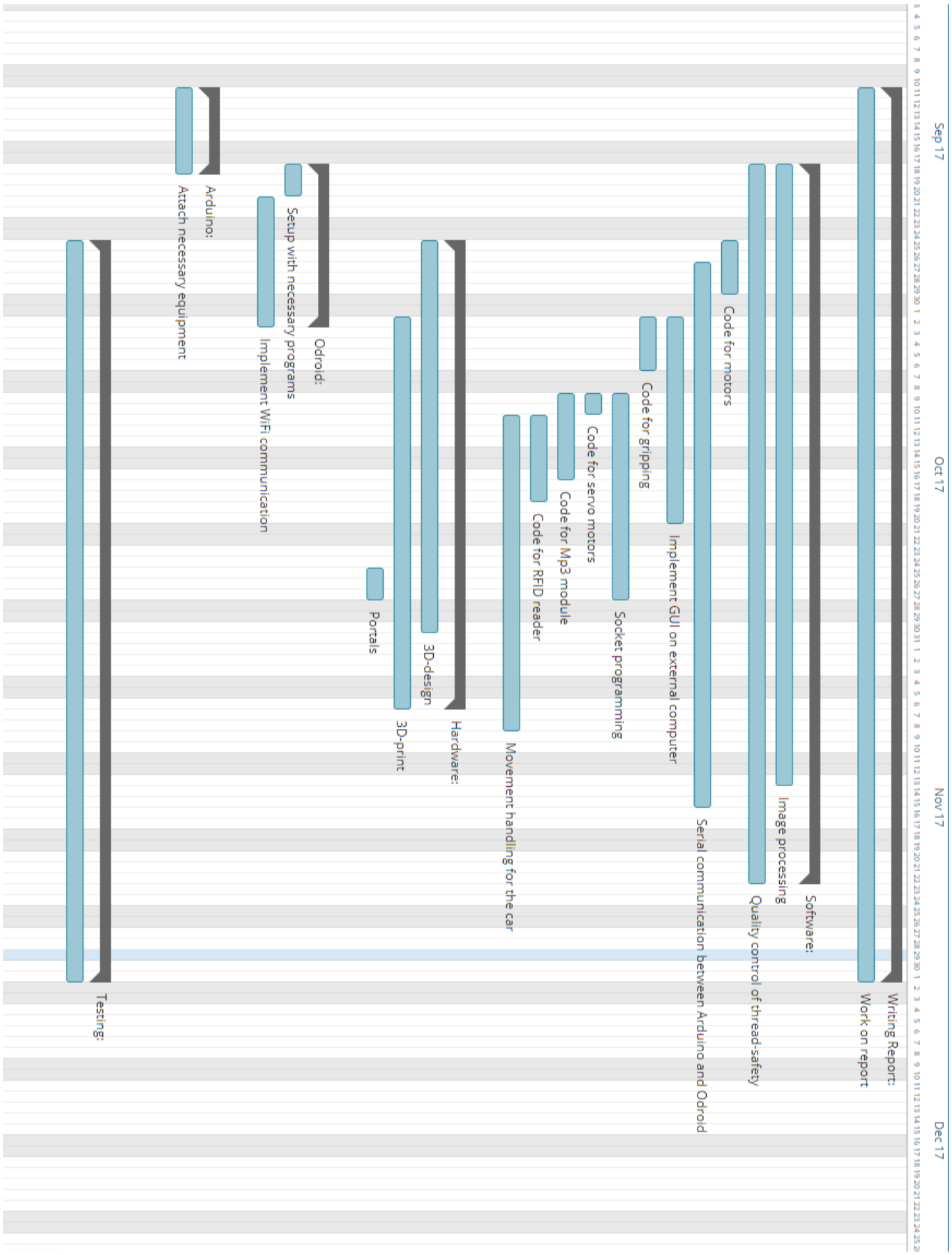
[16] http://openframeworks.cc/ofBook/images/image_processing_computer_vision/images/erosion_in_use.png

[17] https://img1-327a.kxcdn.com/DataImage.ashx/8341485

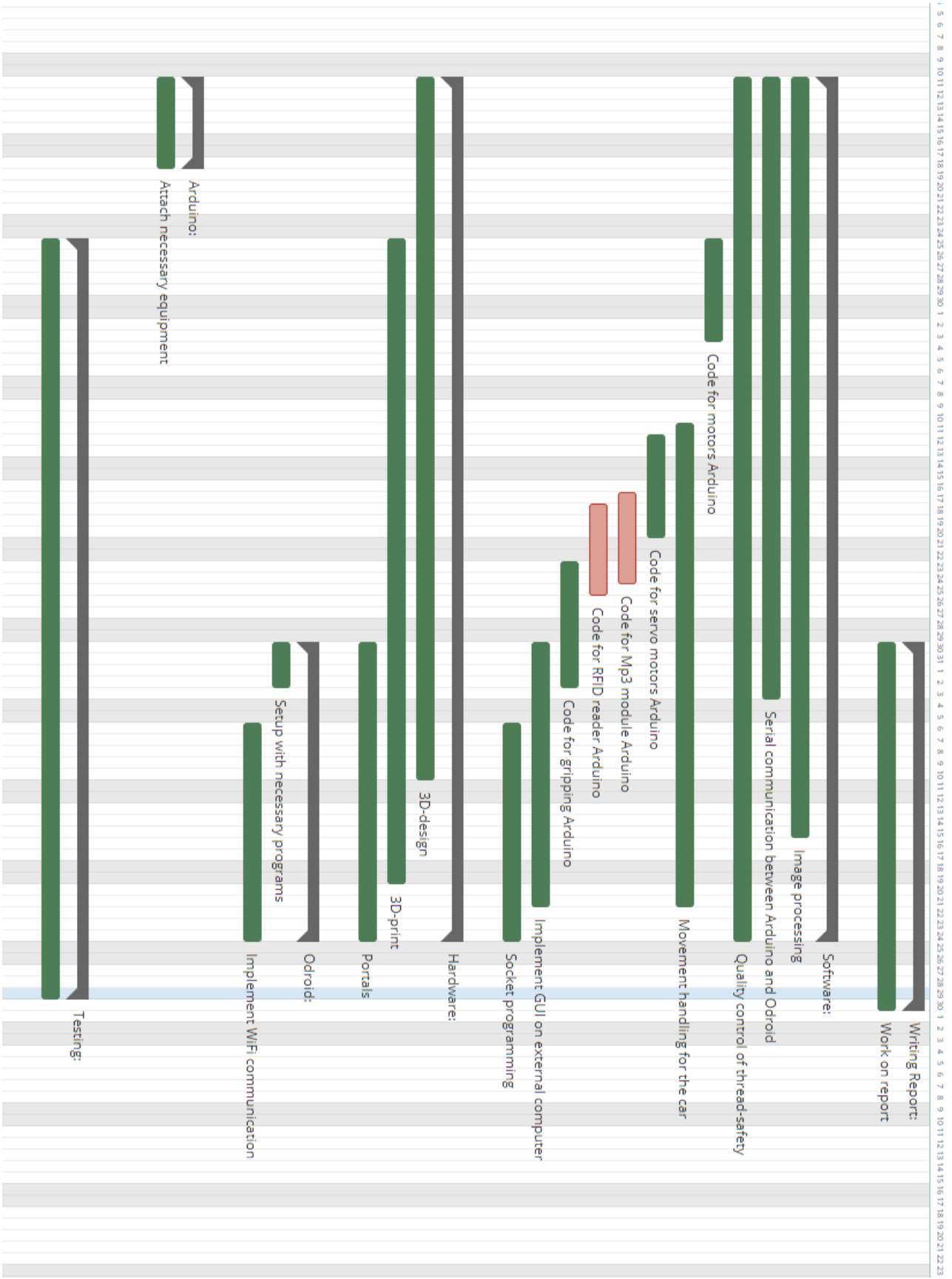
10 APPENDIX

Appendix A	-	Project plan diagram
Appendix B	-	Progress diagram
Appendix C	-	Wiring diagram
Appendix D	-	Java Diagrams (also included in attached folder)
Appendix E	-	Color detection test (video in attached folder)
Appendix F	-	Demonstration (video in attached folder)
Appendix G	-	Source Code for Java and Arduino (in attached folder)

Appendix A – Project plan diagram

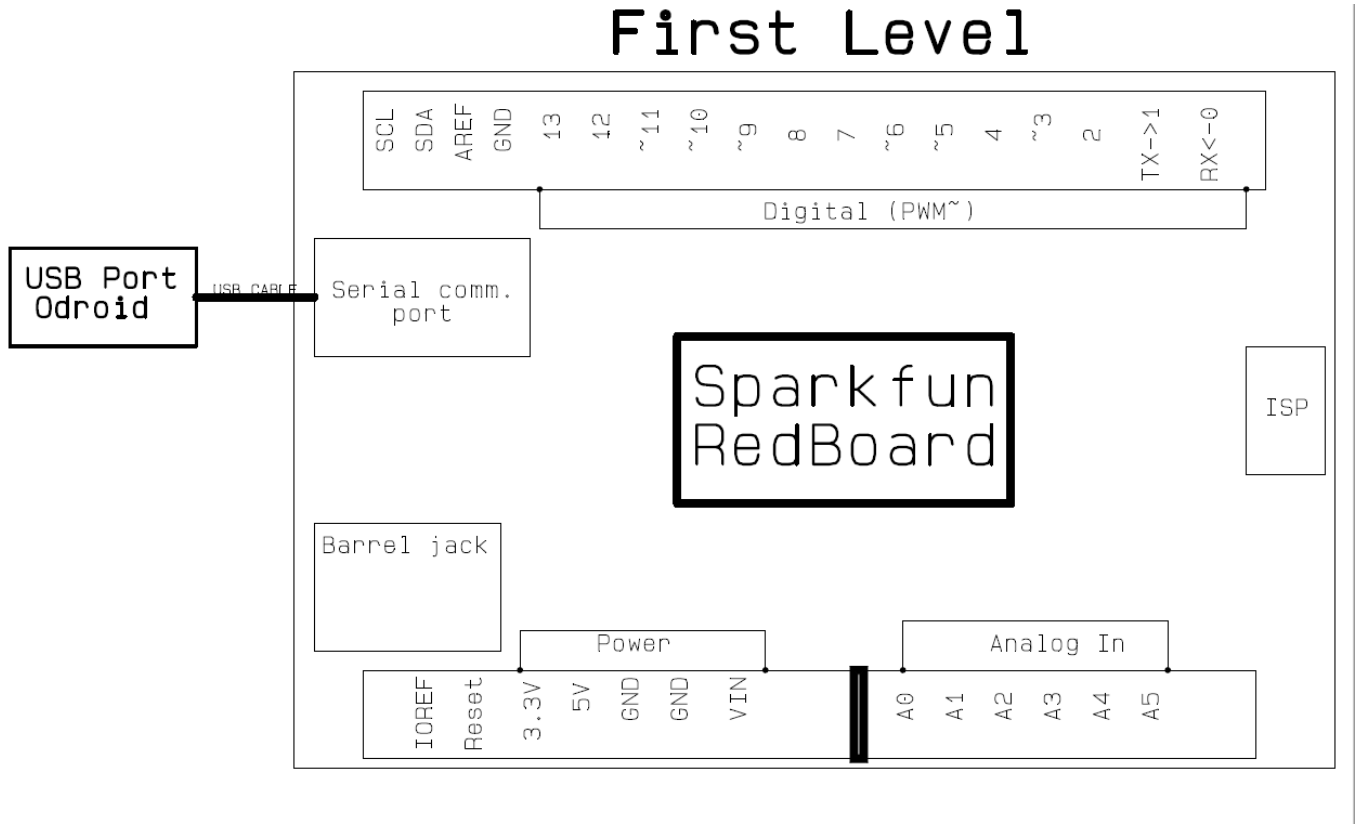


Appendix B – Progress diagram

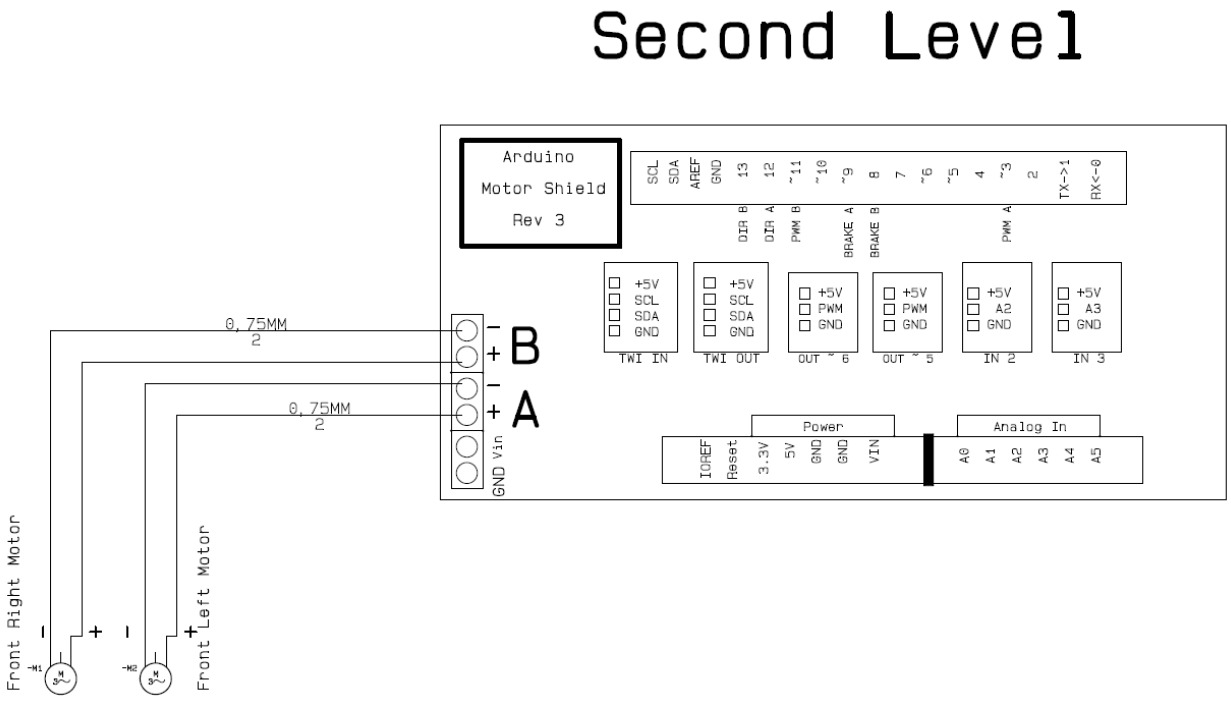


Appendix C – Wiring Diagram

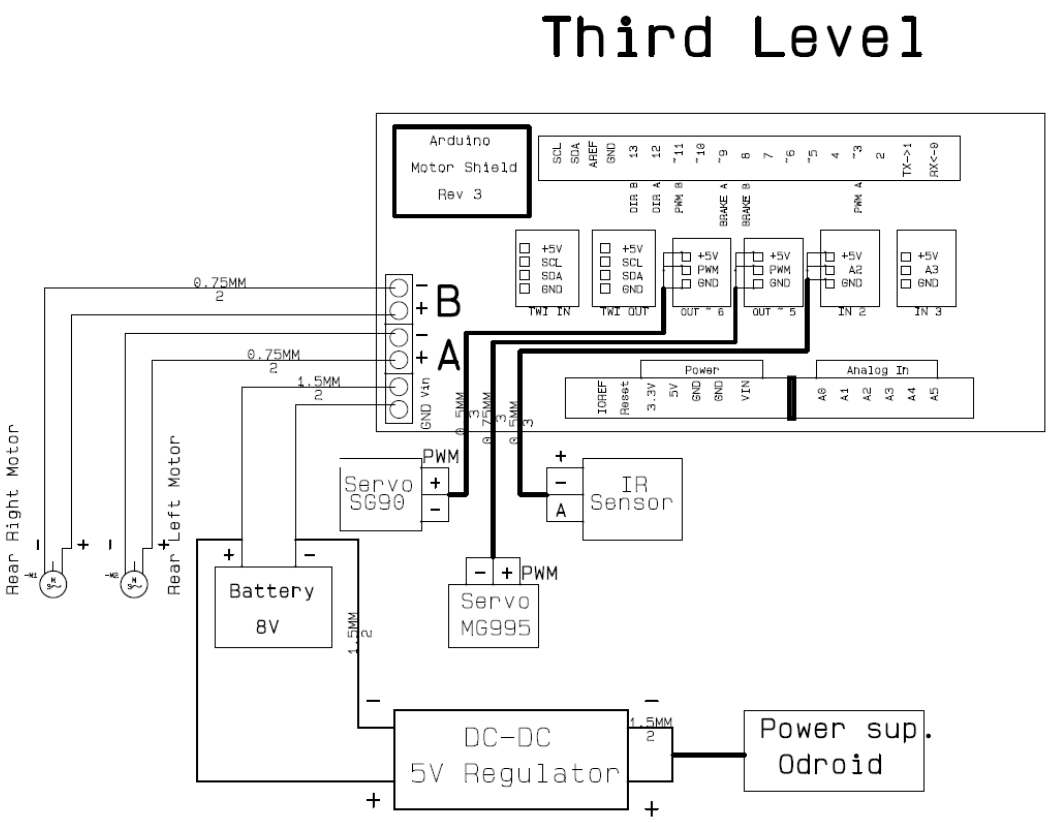
Redboard (Arduino Uno):



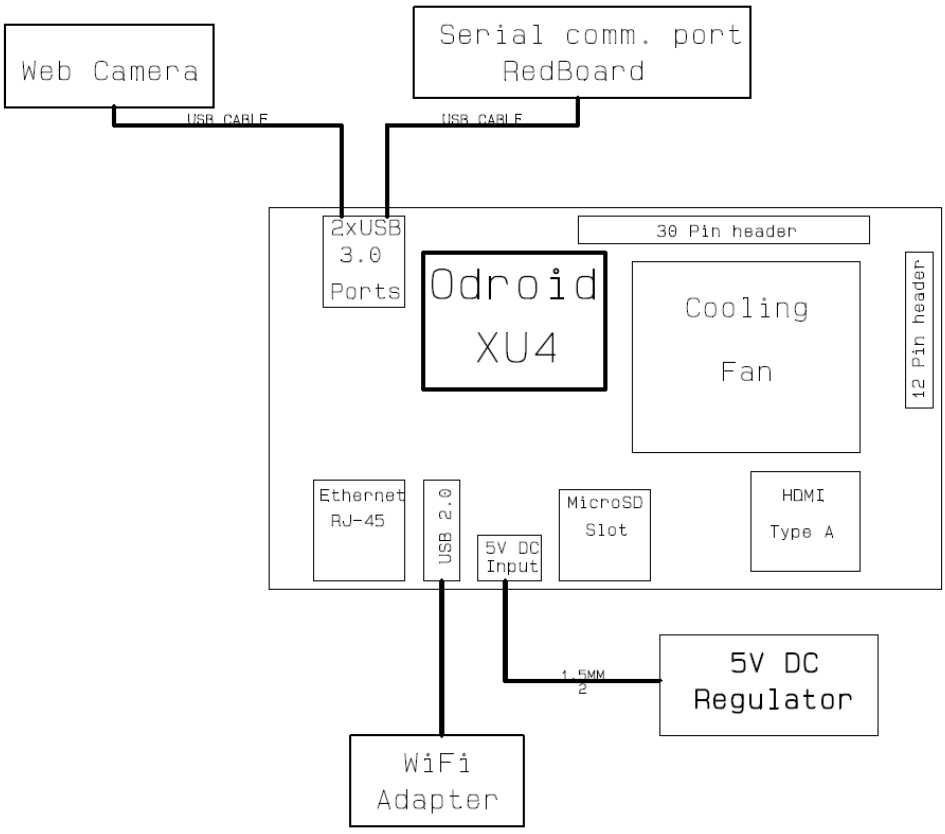
Arduino Motor Shield (first):



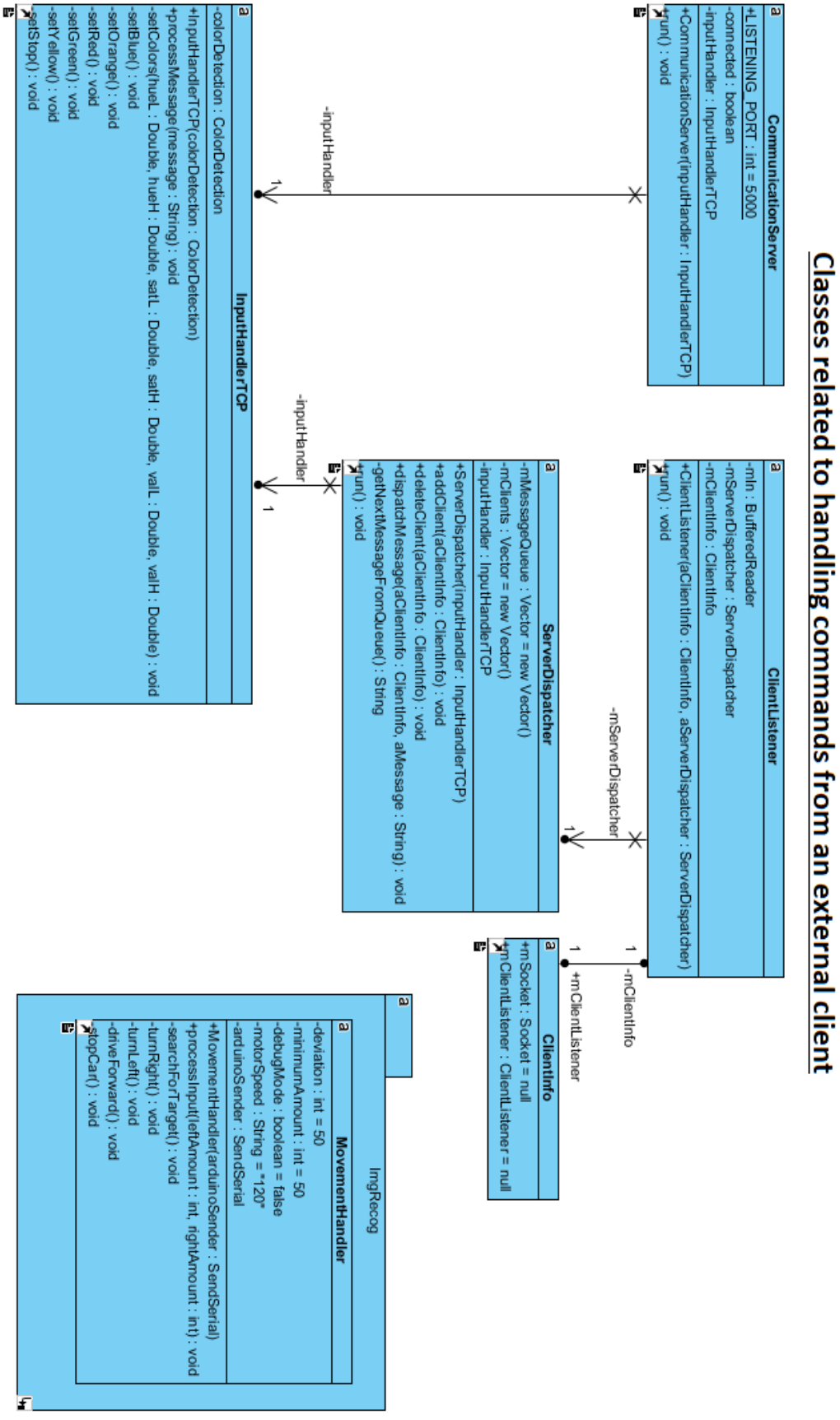
Arduino Motor Shield (second):



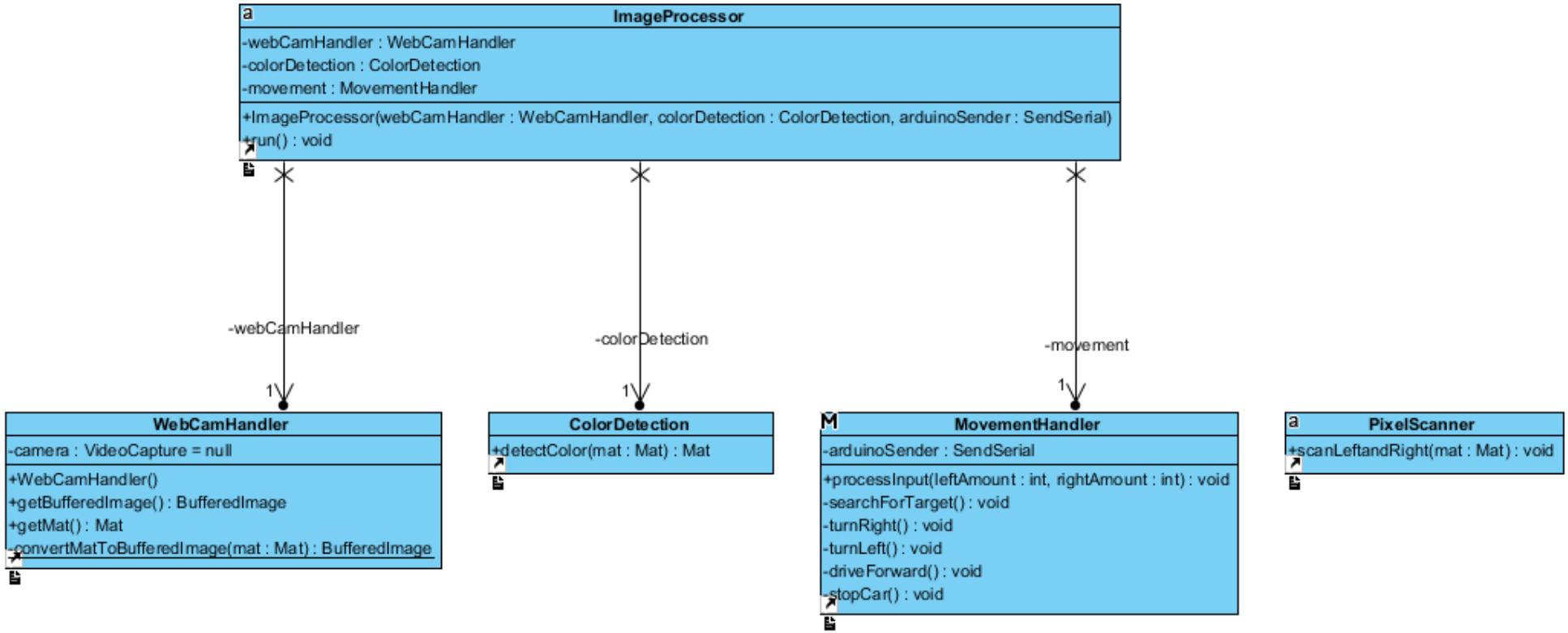
Odroid:



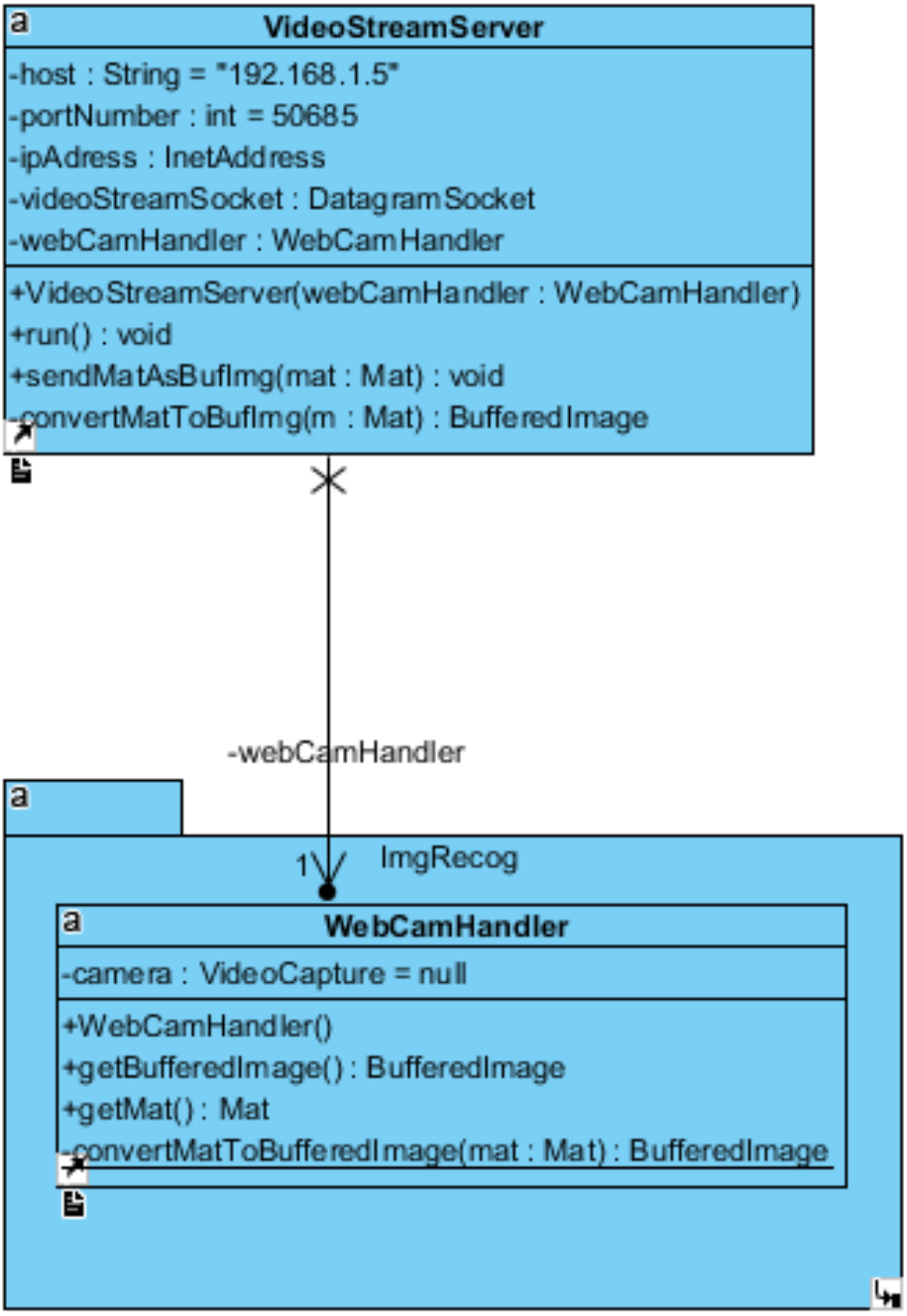
Appendix D – Java Diagrams



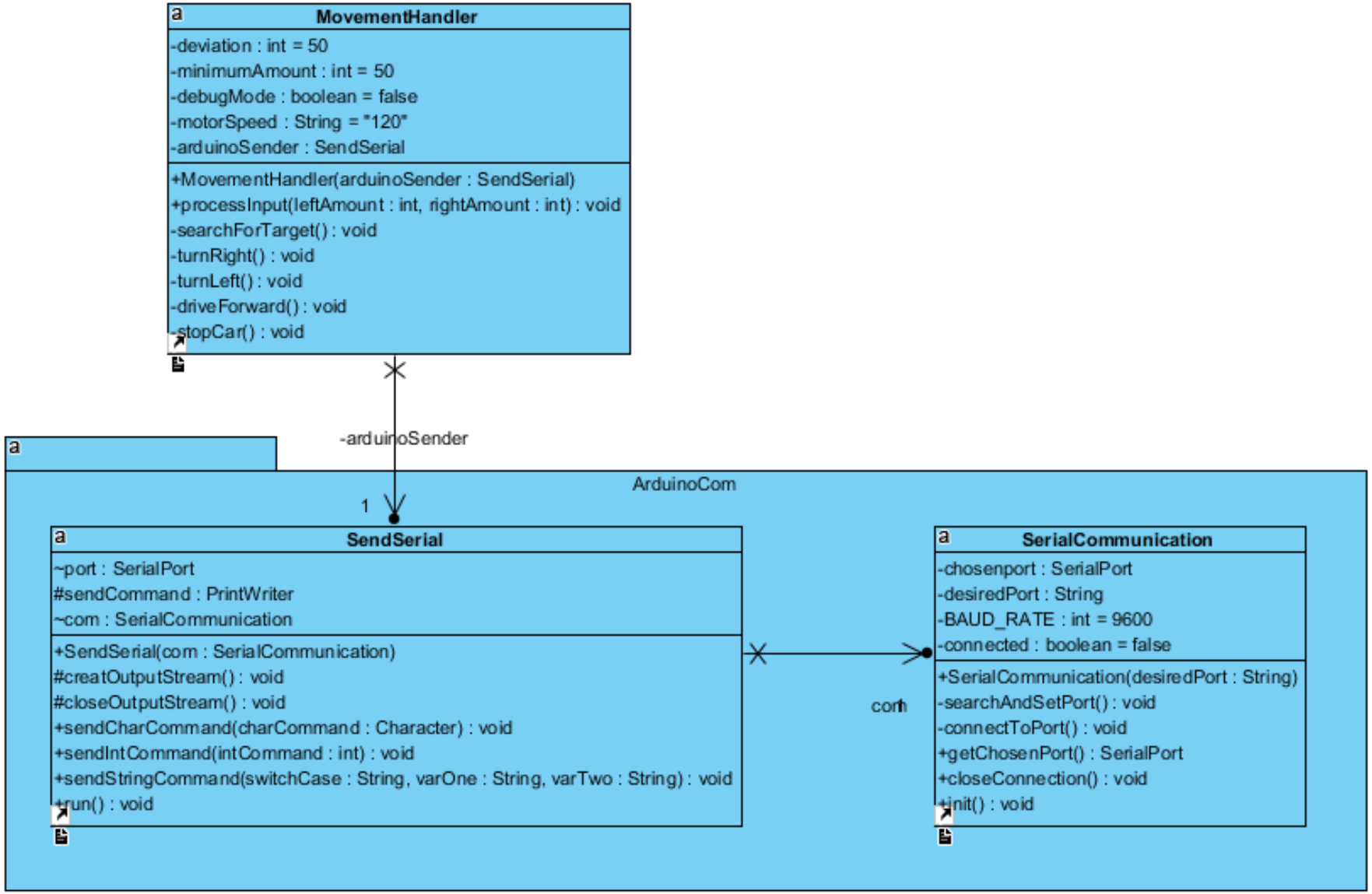
Classes related to image processing



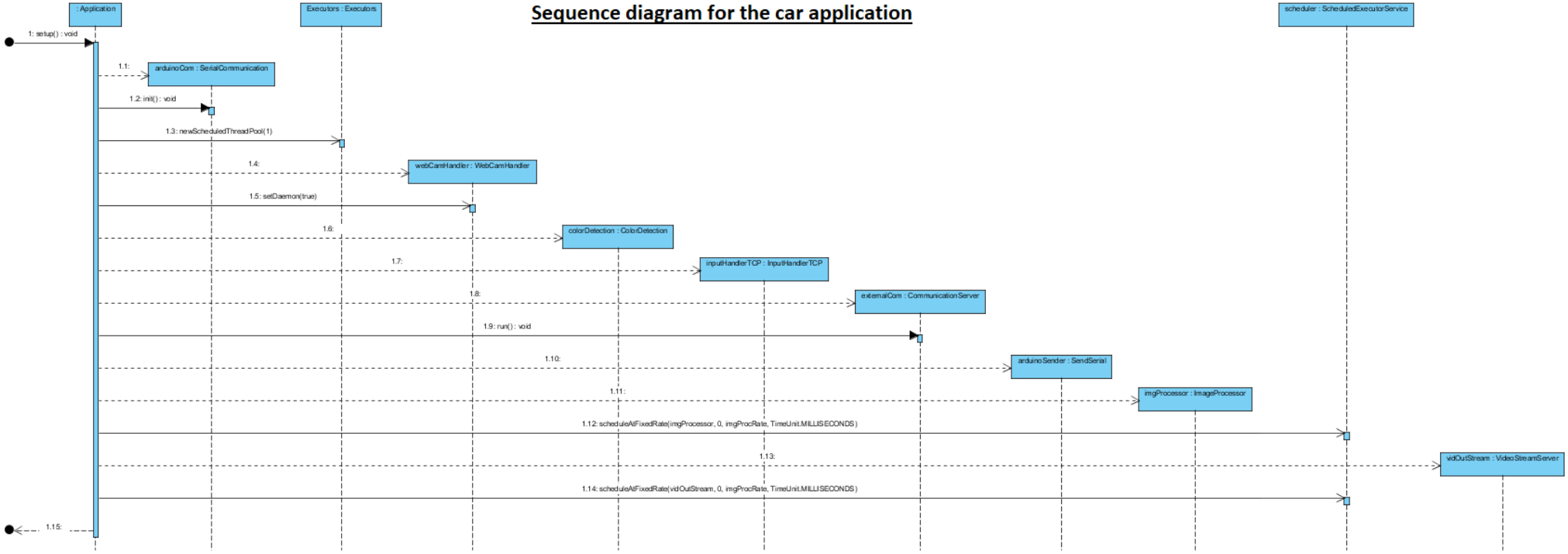
Classes related to streaming video to an external client



Classes related to sending commands to the Arduino



Sequence diagram for the car application



External client (GUI)