**DIPLOMARBEIT**

Gesamtprojekt

**Camera Controlled Swarm Robots**

**Positionserkennung**

Mottl Mario 5BHEL Betreuer: Dipl.-Ing Josef Reisinger

**Swarm Controll**

Clemens Pruggmayer 5BHEL Betreuer: Dipl.-Ing Josef Reisinger

**Visualisierung und Simulation**

Michael reim 5BHEL Betreuer/in: Dipl.-Ing Josef Reisinger

Schuljahr 2020/21

Abgabevermerk:

Datum: TT.MM.JJJJ übernommen von:

****

**Höhere Technische Bundeslehranstalt Hollabrunn**

**Höhere Lehranstalt für Elektronik und Technische Informatik**

**EIDESSTATTLICHE ERKLÄRUNG**

**Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.**

**Mario Mottl**

**Clemens Pruggmayer**

**Michael Reim**

Hollabrunn, am 5. April 2019

**DIPLOMARBEIT**

**DOKUMENTATION**

|  |  |
| --- | --- |
| Namen der  Verfasser/innen | Michael Reim, Clemens Pruggmayer, Mario Mottl |
| Jahrgang  Schuljahr | 5BHEL |
| Thema der Diplomarbeit | Camera Controlled Swarm Robots |
| Kooperationspartner | - |

|  |  |
| --- | --- |
| Aufgabenstellung | Es sollen mehrere 1 bis n viele autonome Fahrzeuge (STM32F107RB + MDDS Board) über eine 1,5m erhöhten Kamera erfasst werden. Positionen der Fahrzeuge sollen in (x/y) Koordinaten verwandelt werden. Diese Daten sollen an die Visualisierung und Simulation geschickt. Diese erzeugt über selbst gezeichnete Bilder einen Weg für die Fahrzeuge. Der Weg soll an das Swarm Controll weitergeschickt werden welche die Positionsdaten mithilfe der Kamera in Fahrkommandos umwandelt. |

|  |  |
| --- | --- |
| Realisierung | Zur Erfassung der Autos wurde eine „DFK 33UX273“ von ImagingSource verwendet. Diese wird per USB an einen Laptop/Computer angeschlossen. Für die Visualisierung und Simulation wurde eine 3D Visualisierungssoftware in C++ geschrieben. Welche die Autos in Echtzeit am Bildschirm anzeigt.  Swarm Controll wurde mithilfe des SvVis Protokoll realisiert. Auf den Autos(Cortex M4) läuft eine selbstgeschriebene RTOS-Software, die die Kommandos des SVIS Protokoll in Bewegungen umwandelt. |

|  |  |  |
| --- | --- | --- |
| Ergebnisse | Eine alte Version des SvVis Protokolls wurde abgeändert und verbessert. Eine Teststrecke wurde aufgebaut und mit einer Halterung für die Kamera erweitert. Positionserkennung wurde mithilfe von Python realisiert. Die Autosteuerungssoftware wurde selbständig mit RTOS realisiert. Visualisierung wurde von Grund auf in C++ + OpenGL geschrieben für den speziellen Anwendungsfall. |  |

|  |  |
| --- | --- |
| Typische Grafik, Foto etc.  (mit Erläuterung) | Die obige Grafik stellt den Aufbau des Gesamtsystems dar. Darin sieht man die 3 Hauptkomponenten Kamera + Kamerasoftware, Visualisierung und Swarm Controll. Daten zwischen den einzelnen Blöcken wir über TCP bereitgestellt. Verbindung zu den Autos kann auf zwei Wege bereitgestellt werden: Bluetooth und Wlan. |

|  |  |
| --- | --- |
| Teilnahme an Wettbewerben,  Auszeichnungen |  |

|  |  |
| --- | --- |
| Möglichkeiten der Einsichtnahme in die Arbeit | HTL Hollabrunn  Anton Ehrenfriedstraße 10  2020 Hollabrunn |

|  |  |  |
| --- | --- | --- |
| Approbation  (Datum / Unterschrift) | Prüfer/Prüferin | Direktor/Direktorin  Abteilungsvorstand/Abteilungsvorständin |

**DIPLOMA THESIS**

**Documentation**

|  |  |
| --- | --- |
| Author(s) | Michael Reim, Clemens Pruggmayer, Mario Mottl |
| Form  Academic year | 5BHEL |
| Topic | Camera Controlled Swarm Robots |
| Co-operation partners | - |

|  |  |
| --- | --- |
| Assignment of tasks | One or more autonomous vehicle (powered by an STM32F107RB + MDDS Board) should get detected by a camera that is attached 1,5 metres above the table. The positions of the vehicles should be converted into an x-y coordinate grid. The produced data should then be sent to our visualisation and simulation for correction purposes. A picture of a path should be drawn. The generated path should then be sent off to the Swarm Control. Where the path will be transformed into vehicle commandos. |

|  |  |  |
| --- | --- | --- |
| Realisation | For proper detection, an “DFK 33UX273” from Imaging Source was used. It is connected via USB to a Laptop. The visualisation and simulation were programmed in “C++” + “OpenGl” which draws the cars onto the screen in Realtime. Swarm Control uses a technology called “SvVis”. The vehicles run on a self-implemented RTOS-Software (Real Time Operating System) which transforms the vehicle commandos into engine movement. |  |

|  |  |
| --- | --- |
| Results | An older version of the “SvVis” was used and altered to fit our purposes. A test track with a bracket for the camera was built. The tracking software was programmed in python. Vehicle Control software was written with RTOS. The visualisation and simulation were programmed in “C++” + “OpenGl” for out specific purpose. |

|  |  |
| --- | --- |
| Illustrative graph, photo  (incl. explanation) | The graph above pictures the whole System. There you can see the three main components. Tracking, Swarm Control and Visualisation and Simulation. The communication between the individual blocks is realised in TCP. The communication between laptop and cars can be established in two ways WLAN, Bluetooth. |

|  |  |
| --- | --- |
| Participation in competitions  Awards | - |

|  |  |
| --- | --- |
| Accessibility of  final project thesis | HTL Hollabrunn  Anton Ehrenfriedstraße 10  2020 Hollabrunn |

|  |  |  |
| --- | --- | --- |
| Approval  (Date / Signature) | Examiner/s | Head of Department / College |

DA Antrag und unterschriebene Erklärung aus der Diplomarbeitsdatenbank einfügen

Inhaltsverzeichnis

[1 Camera Controlled Swarm Robots 12](#_Toc64022130)

[1.1 Blockschaltbild 12](#_Toc64022131)

[1.2 Funktionsbeschreibung 12](#_Toc64022132)

[2 Positionserkennung 13](#_Toc64022133)

[2.1 Übersicht 13](#_Toc64022134)

[2.2 Aufbau 13](#_Toc64022135)

[2.3 Kamera Ansteuerung 13](#_Toc64022136)

[2.4 Erkennung Positions LEDs 13](#_Toc64022137)

[2.4.1 Algorithmus zur Erkennung 14](#_Toc64022138)

[2.4.2 Berechnung der Position 15](#_Toc64022139)

[2.5 Auswertung der Bilddaten 15](#_Toc64022140)

[2.6 Kommunikation mit Simulation / Visualisierung 15](#_Toc64022141)

[3 Visualisierung und Simulation 17](#_Toc64022142)

[3.1 Übersicht Softwarearchitektur 17](#_Toc64022143)

[3.2 Aufbau der Engine 17](#_Toc64022144)

[3.3 Erstellung von 3D-Modellen 17](#_Toc64022145)

[3.4 Bewegungen im Dreidimensionalen Raum 17](#_Toc64022146)

[3.5 Auswertung von erhaltenen Positionsdaten 17](#_Toc64022147)

[3.6 Übertragung zu Swarm Controll 17](#_Toc64022148)

[4 Swarm Controll 18](#_Toc64022149)

[4.1 Übersicht Softwarearchitektur 18](#_Toc64022150)

[4.2 Berechnung des Weges 18](#_Toc64022151)

[4.3 Konvertierung von Positionen auf Steuerbefehle (WIP) 18](#_Toc64022152)

[4.4 Steuerbefehle 18](#_Toc64022153)

[4.4.1 Aufbau von Steuerbefehlen (Protokollaufbau) 18](#_Toc64022154)

[4.4.2 Senden von Steuerbefehlen (API) 18](#_Toc64022155)

[5 Fahrzeug Software 19](#_Toc64022156)

[5.1 Übersicht der Architektur 19](#_Toc64022157)

[5.2 Kommunikation mit Swarm Controll 19](#_Toc64022158)

[5.2.1 Kommunikationsprotokoll 19](#_Toc64022159)

[5.3 Hardware Ansteuerung 19](#_Toc64022160)

[5.3.1 LED-Ansteuerung 19](#_Toc64022161)

[5.3.2 Kommunikationsmodule 19](#_Toc64022162)

[5.3.2.1 Bluetooth 19](#_Toc64022163)

[5.3.2.2 WLAN 19](#_Toc64022164)

[5.3.3 Motor Ansteuerung 19](#_Toc64022165)

[6 Ergebnisse 20](#_Toc64022166)

[6.1 Funktionalität Positionserkennung 20](#_Toc64022167)

[6.2 Steuerung der Fahrzeuge 20](#_Toc64022168)

[6.3 Simulationstest mit Pseudodaten 20](#_Toc64022169)

[6.4 Steuersoftwarte Funktionalitätstest 20](#_Toc64022170)

[6.5 Schwarmbewegung 20](#_Toc64022171)

[6.5.1 Kreis 20](#_Toc64022172)

[6.5.2 Zick-Zack 20](#_Toc64022173)

[7 Anhang 21](#_Toc64022174)

[7.1 Inbetriebnahme (f. 4Klasse TdoT) 21](#_Toc64022175)

[7.2 Projektmanagement 21](#_Toc64022176)

[7.2.1 Projektplan 21](#_Toc64022177)

[7.2.2 Projekttagebuch 21](#_Toc64022178)

[7.2.3 Projektkosten 21](#_Toc64022179)

[8 Quellenverzeichnis 22](#_Toc64022180)

[8.1 Bücher 22](#_Toc64022181)

[8.2 Onlinemedien 22](#_Toc64022182)

[8.3 Zeitschriften 22](#_Toc64022183)

Wenn Sie in Ihrer Dokumentation ausschließlich mit den „Formatvorlagen“ dieses Dokuments gearbeitet haben können Sie hier sehr einfach das Inhaltsverzeichnis automatisch aktualisieren lassen.

Klicken Sie dazu mit der rechten Maustaste in das vorhandene Inhaltsverzeichnis und wählen Sie im Kontextmenü „Felder aktualisieren“ und dann „gesamtes Verzeichnis aktualisieren“ auswählen – fertig! … diesen Text löschen!

# Camera Controlled Swarm Robots

## Blockschaltbild

## Funktionsbeschreibung

# Positionserkennung

## Übersicht

## Aufbau

## Kamera Ansteuerung

## Erkennung Positions LEDs

Für die Erkennung der Positions LEDs wird ein simpler Algorithmus verwendet. Durch die spezielle Anordnung der LEDs in einem gleichschenkeligen Dreieck kann man die ungefähre Position der anderen LEDs mit dem Satz des Pythagoras berechnen.



Um sicherzustellen, dass die Positions LEDs zu einem Auto und nicht zu einem anderen gehören. Werden immer nur zwei LEDs mit der Kamera erkannt und die Position der letzten LED wird berechnet.

### Algorithmus zur Erkennung

Für den Algorithmus wird der in OpenCV eingebaute Filter + Detektor verwendet. Die Parameter werden wie folgt gesetzt.

params = cv2.SimpleBlobDetector\_Params()

# Change thresholds

params.minThreshold = 0

params.maxThreshold = 256

# Filter by Area.

params.filterByArea = True

params.minArea = 30

# Filter by Circularity

params.filterByCircularity = True

params.minCircularity = 0.1

# Filter by Convexity

params.filterByConvexity = True

params.minConvexity = 0.5

# Filter by Inertia

params.filterByInertia = True

params.minInertiaRatio = 0.5

detector = cv2.SimpleBlobDetector\_create(params)

Diese Klasse ist besonders gut geeignet um blobs (zusammenhängende Bildpunkte mit derselben Farbe und Intensität). Diese Klasse implementiert einfache Algorithmen, um diese zu extrahieren.

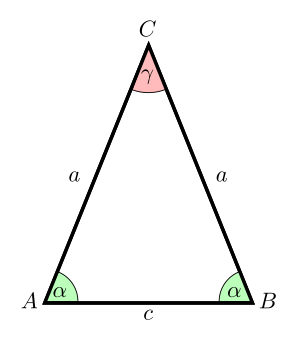
Konvertierung des Quellbildes in Binärbild, indem man einen Schwellenwert mit mehreren  
Schwellenwerten von minThreshold (einschließlich) bis maxThreshold (exklusiv) mit einem Abstand Threshholdstep zwischen benachbarten Schwellenwerten anwendet.

Extrahieren von verbundenen Komponenten aus jenem Binärbild mit findContours und berechnen von deren Zentren.

Gruppierung von Zentren aus mehreren Binärbildern anhand ihrer Koordinaten. Nahe Zentren bilden eine Gruppe, die einem Blob entspricht, der vom Parameter minDistBetweenBlobs gesteuert wird.

Schätzen aus den Gruppen die endgültigen Zentren der Blobs und ihre Radien und geben diese als Positionen und Größen der Schlüsselpunkte zurück.

### Berechnung der Position



Um die Seitenlängen zu bestimmen wurde einmal als Referenz a & c in Pixel gemessen.

Es wird weiteres als Kontrolle der Winkel α & γ berechnet.

Darüber hinaus wird in regelmäßigen Abständen die Seitenlänge neu berechnet. Um sicherzustellen, dass es sich noch immer um dasselbe Auto handelt.

## Auswertung der Bilddaten

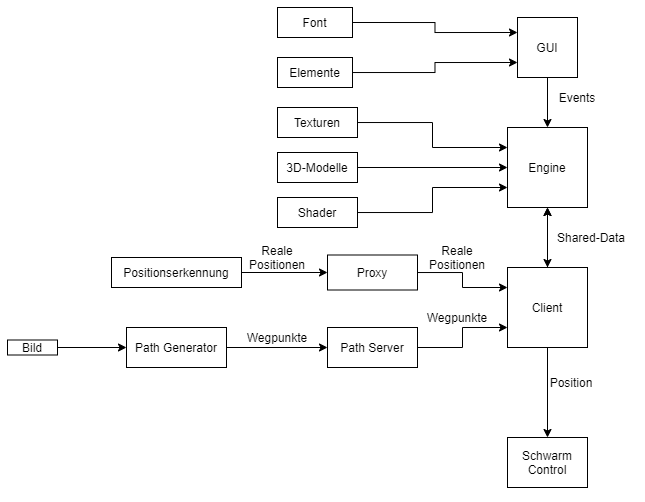
Um aus den vielen Bilddaten eine vernünftige Position zu generieren wird der Mittelpunkt des Autos berechnet. Jede Koordinate einer Positions LED wird in einem Tupel (x, y) (posLed) abgespeichert. Jedes Auto (cararray) besteht aus einem Array aus drei Tupel (posLed1, posLed2, posLed3) + Fahrtrichtung. Über diese drei Tupel wird gemittelt, um den Mittelpunkt des Fahrzeugs zu erhalten.

## Kommunikation mit Simulation / Visualisierung

# Visualisierung und Simulation

## Übersicht Softwarearchitektur

### Blockschaltbild



### GUI

Das GUI ist die graphische Oberfläche der Visualisierung. Es enthält Text und verschiedene Elemente wie zum Beispiel eine Eingabezeile. Die Schriftart, die verwendet wird, wird über ein File in das Programm eingelesen. Mehr zu diesem Thema gibt es im Kapitel 3.2 (Graphical User Interface).

### Engine

Die Engine ist das Rückardt der Software. Sie beinhaltet das graphische Rendering, steuert Bewegungen und den Ablauf der Simulation und reagiert auf Events. Dazu wird im Kapitel 3.3 (Aufbau der Engine) näher eingegangen.

### Client

Der Client ist für das Empfangen und Senden von Daten über das Netzwerk zuständig. Da die Engine mit dem Client kommunizieren muss, wird ein Shared-memory verwendet. Auf diesen Speicher haben nur die Engine und der Client Zugriff.

### Modelle, Texturen, Shader

Modelle beschreiben die Form der Objekte in einer Rendering-Szene. Sie beinhalten Raumpunkte, genannt Vertices, Normalvektoren und Texturkoordinaten.  
Texturen werden verwendet, um Modellen Farben zu geben. Diese sind Bild-Files, welche in den Speicher geladen werden, und anschließend am Modell angezeigt werden. Welcher Bereich einer Textur auf einem Modell angezeigt wird, wird über die Texturkoordinaten gesteuert.

Shader sind Programme, die auf der Graphikkarte ausgeführt werden. Sie berechnen mithilfe der Vertices, Texturkoordinaten und Normalvektoren die endgültige Farbe eines Pixels auf dem Bildschirm. Dieser Prozess wird „Shading“ genannt. Genauer erläutert wird das im Kapitel 3.3 (Aufbau der Engine) und 3.4 (Erstellung von 3D-Modellen).

### Path Generierung

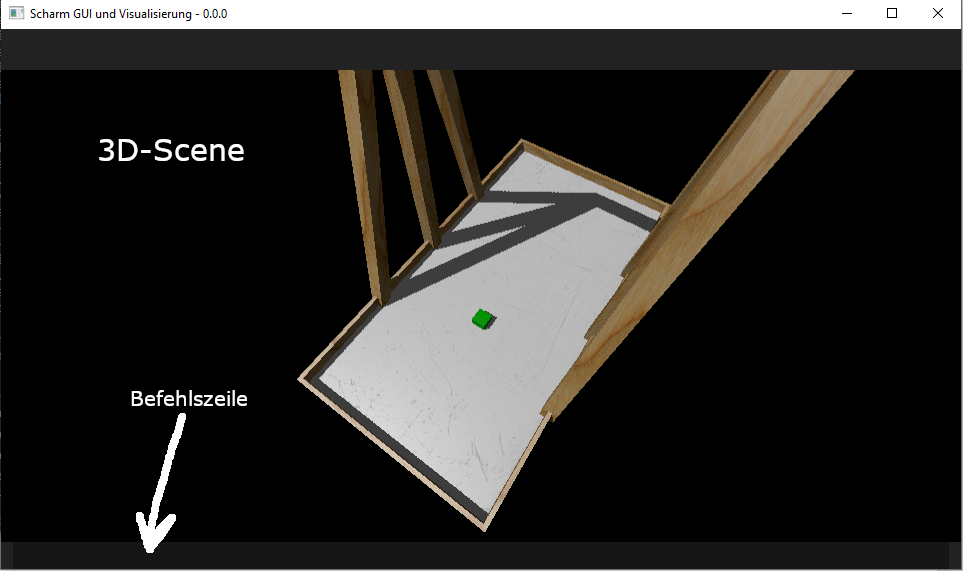
Die Generierung eines Weges erfolgt aus einem Bild-File. Der Anwender kann mit einem Programm, womit man Bild-Files editieren kann, einen Beliebigen weg auf das Bild zeichnen. Dieses Bild wir anschließend vom „Path Generator“ eingelesen und in Wegpunkte untereilt, welche die Fahrzeuge abfahren. Diese Wegpunkte und dessen Nummer erhält der „Path Server“. Er ist für die Aufbereitung und für das Senden der Wegpunkte zur Visualisierung zuständig.  
Wenn die realen Fahrzeuge angesteuert werden, werden neben den generierten Wegpunkten, noch zusätzliche Positionen von der Positionserkennung empfangen. Diese Positionen sind die Koordinaten der abgefilmten Fahrzeuge und werden verwendet, um die Bahnen der virtuellen Fahrzeuge in der Visualisierung zu korrigieren.

Da die Positionserkennung Daten in der falschen Byteorder sendet, ist ein Proxy zwischen der Positionserkennung und der Visualisierung, der die Byteorder umdreht und weitersendet.

Genauer wird im Kapitel 3.6 (Auswertung von erhaltenen Positionsdaten) auf dieses Thema eingegangen.

## Graphical User Interface

### Aufbau



**3D-Scene:** In der 3D-Scene wird zum Anzeigen der Visualisierung und Simulation verwendet. In ihr werden die Fahrzeuge und das Modell des Tisches gerendert.

**Befehlszeile:** Über die Befehlszeile werden die Fahrzeuge im realen Aufbau, sowie in der Simulation, gesteuert.

Anmerkung: Die Befehle zur Ansteuerung des realen Aufbaus wurden aus Zeitgründen noch nicht realisiert!

### Befehle

Jeder Befehl, der in die Befehlszeile eingegeben wird, muss mit einem „/“ beginnen.

#### Befehle für die Simulation

Syntax: /simu generate <image-file> <vehicle-ID> <number of goals>

1. generate  
   Dieses Argument gibt der Simulation die Anweisung, dass ein neuer weg für ein Fahrzeug generiert werden soll.
2. <image-file>  
   Typ: string  
   Gibt an welche Bilddatei eingelesen wird. Dabei muss beachtet werden, dass ein Standardverzeichnis angegeben werden kann. Wenn ein Standardverzeichnis verwendet wird, muss der Dateipfad relativ zu dem Standardverzeichnis angegeben werden!
3. <vehicle-ID>  
   Typ: unsigned integer  
   Gibt an für welches Fahrzeugt der Weg generiert werden soll. Es ist zu beachten, dass jedes Fahrzeug eine numerische ID, beginnend bei 0 (Typ: unsigned integer), besitzt.
4. <number of goals>  
   Typ: unsigned integer  
   Gibt die Auflösung des Weges an. Der Weg wird in einzelne Punkte unterteilt, die vom Fahrzeug nacheinander abgefahren werden. Desto mehr Punkte (Goals) generiert werden, umso genauer wird der weg in der Simulation nachgefahren.

Nach Ausführung dieses Befehls, wird der eingegebenen Fahrzeugnummer, der generierte Weg zugeordnet.

Syntax: /simu start

Nach Ausführung dieses Befehls wird der Simulationsprozess eingeleitet. Der Simulationsprozess beendet sich von allein, wenn alle Fahrzeuge ihren Weg nachgefahren sind.

Syntax: /simu stop

Nach Ausführung dieses Befehls wird der Simulationsprozess unterbrochen.

Syntax: /simu reset

Dieser Befehl löscht bei allen Fahrzeugen den zugehörigen Weg.

Achtung: Dieser Befehl kann nur ausgeführt werden, wenn die Simulation beendet ist. Um die Simulation abzubrechen nutze: „/simu stop“.

#### Befehle für den realen Aufbau

NOCH NICHT REALISIERT!

### Realisierung

#### Erstellung und Initialisierung vom GUI

/\* CREATE TEXTBOXES \*/

    std::cout << get\_msg("INFO / GUI") << "Creating textboxes..." << std::endl;

    GUI::TextBox cmd\_line(main\_window\_handler.get\_width\_ptr(), main\_window\_handler.get\_height\_ptr(), main\_window\_handler.get\_aspect\_ptr());

    cmd\_line.set\_pos(-0.975f, -1.0f);

    cmd\_line.set\_size(1.95f, 0.1f);

    cmd\_line.set\_textinput\_color(Main::TEXTBOX\_ACTIVE\_COLOR);

    cmd\_line.set\_cursor\_color(Main::TEXTBOX\_CURSOR\_COLOR);

    cmd\_line.set\_font\_size(0.075f);

    cmd\_line.set\_text\_color(Main::TEXTBOX\_ACTIVE\_TEXT\_COLOR);

    std::cout << get\_msg("INFO / GUI") << "Textboxes created..." << std::endl;

Hier wird die Kommandozeile für die Befehlseingebe erstellt. Es werden einige Parameter, wie Höhe, Breite, Schriftgröße und Textfarbe gesetzt.

/\* -----------------------------------------------------------------------------

     \* INIT ELEMENT HANDLER

     \* -----------------------------------------------------------------------------\*/

    GUI::ElementHandler element\_handler;

    element\_handler.attach\_element(cmd\_line);

    element\_handler.start(std::chrono::milliseconds(5));

    std::cout << get\_msg("INFO / GUI") << "Started Element-Handler." << std::endl;

Der Element-Handler steuert die ganzen Interaktionen der übergebenen GUI-Bausteine, wie Buttons, Textzeilen, usw. Darüber hinaus ruft er Events auf, sodass der Event-Handler darauf reagieren kann. Der Vorteil ist, dass der Event-Handler frei programmierbar ist und der Programmierer bestimmen kann, was passiert, wenn er mit den GUI-Bausteinen interagiert.

/\* -----------------------------------------------------------------------------

     \* INIT ELEMENT RENDERER

     \* -----------------------------------------------------------------------------\*/

    GUI::ElementRenderer element\_renderer(context, cmd\_queue);

    element\_renderer.attach\_handler(element\_handler);

    element\_renderer.init();

    element\_renderer.start(std::chrono::milliseconds(5));

    std::cout << get\_msg("INFO / GUI") << "Started Element-Renderer." << std::endl;

    std::cout << get\_msg("INFO / GUI") << "Successfully initialized GUI." << std::endl;

Der Element-Renderer generiert die graphischen Daten, auch Meshes genannt, sodass man das Userinterface mit OpenGL rendern kann.

#### Erstellung der Events für das GUI

class TextInpListener : public Listener

{

private:

    TextBoxEnterEvent textboxenter;

    inline static Schwarm::Client::SharedSimulationMemory\* sharedsimumem;

protected:

    virtual void init(void)

    {

        this->register\_event(textboxenter, reinterpret\_cast<EventFunc>(on\_textenter));

        this->set\_interval(std::chrono::milliseconds(5));

    }

public:

    TextInpListener(void)

    {

        this->init();

    }

    static void set\_sharedsimumem(Schwarm::Client::SharedSimulationMemory\* mem)

    {

        sharedsimumem = mem;

    }

    static void on\_textenter(TextBoxEnterEvent& event)

    {

        std::vector<std::string> args;

        if(Schwarm::decode\_command(event.get\_text\_input().get\_text\_value(), args))

            Schwarm::on\_command(args, sharedsimumem);

        else

            std::cout << get\_msg("ERROR") << "A command has to begin with \'/\'." << std::endl;

        event.get\_text\_input().set\_text\_value("");

    }

};

Die Methode „on\_textenter(TextBoxEnterEvent& event)” wird aufgerufen, nachdem ein Befehl mit ENTER bestätigt wurde. Hier wird der Befehl mit der Funktion „decode\_command“ in seine einzelnen Argumente dekodiert. In der „on\_command“ Funktion wird auf die im Punkt **3.2.2** genannten Befehle reagiert und die entsprechende Operation ausgeführt.

## Aufbau der Engine

### Blockschaltbild

### Graphisches Rendering

Als Graphik API wird OpenGL verwendet.

#### Texturen, Modelle und Shader

gl::Model table;

gl::model\_error\_t model\_error = table.load("Models/", "table.obj", gl::Model::MESH\_COMBINED | gl::Model::INVERT\_T);

gl::Model vehicle;

model\_error = vehicle.load("Models/", "vehicle.obj", gl::Model::MESH\_COMBINED | gl::Model::INVERT\_T);

const gl::MeshMatierial table\_mtl = table.get\_materials().begin()->second;         // there is only one material

const gl::MeshMatierial vehicle\_mtl = vehicle.get\_materials().begin()->second;

if(model\_error != gl::model\_error\_t::MODEL\_ERROR\_NONE)

{

    return -1;

}

Hier wird das Modell des Tisches (table.obj) und der Fahrzeuge (vehicle.obj) geladen.

Das 3D-Modell ist in einer Object-Datei (.obj) abgespeichert und wird mit einer Library ausgelesen. Anschließend werden die Modelle zu den sogenannten Meshes konvertiert, um sie am Bildschirm anzeigen zu können.

void load\_texture(unsigned int& texture, const std::string path, bool use\_gamma)

{

    std::cout << get\_msg("INFO / TEXTURE") << "Loading texture \"" << path << "\"..." << std::endl;

    int x\_pixels, y\_pixels;

    uint8\_t\* data = stbi\_load(path.c\_str(), &x\_pixels, &y\_pixels, NULL, 4);  // always load 4 components because the internal format requieres RGBA

    if(data == NULL)

    {

        std::cout << get\_msg("ERROR / TEXTURE") << "Unable to load texture \"" << path << "\"." << std::endl;

        exit(-1);

    }

    else

    {

        glGenTextures(1, &texture);

        glBindTexture(GL\_TEXTURE\_2D, texture);

        glTexImage2D(GL\_TEXTURE\_2D, 0, (use\_gamma) ? GL\_SRGB\_ALPHA : GL\_RGBA, x\_pixels, y\_pixels, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, data);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glBindTexture(GL\_TEXTURE\_2D, 0);

        stbi\_image\_free(data);

        std::cout << get\_msg("INFO / TEXTURE") << "Texture \"" << path << "\" loaded." << std::endl;

    }

}

Hier werden die Texturen mit der Funktion „load\_texture“ geladen. Texturen sind Bilddateien, die für die komplexe Färbung von Objekten verwendet werden. In diesem Codeabschnitt wird eine Bilddatei, mit der stb\_master library, in den Speicher eingelesen. Anschließend wird ein neuer Texturpuffer mit „glGenTextures“ auf der Grafikkarte im VRAM erstellt. Zum Schluss wird das eingelesene Bild mit „glTexImage2D“ an die Grafikkarte gesendet und ist nun im VRAM gespeichert. Mit „glTexParameteri“ werden Filteroperationen für Texturen gesetzt. Diese Filteroperationen werden intern von der Grafikkarte berechnet.

void load\_shader(gl::Shader& shader, const std::string& vertex\_fname, const std::string& fragment\_fname)

{

    static const std::string path("Shaders/");

    std::cout << get\_msg("INFO / LOADING SHADER") << "Loading shader \"" << path + vertex\_fname << "\" and \"" << path + fragment\_fname << "\"..." << std::endl;

    gl::ShaderLoadError err = shader.load(path + vertex\_fname, path + fragment\_fname);

    if(err & gl::ShaderLoadErrorType::INVALID\_FILE\_PATH)

    {

        std::cout << get\_msg("ERROR / LOADING SHADER") << "Unable to find path \"" << path + vertex\_fname << "\" or \"" << path + fragment\_fname << "\"." << std::endl;

        exit(-1);

    }

    if(err & gl::ShaderLoadErrorType::SHADER\_ALREADY\_LOADED)

    {

        std::cout << get\_msg("ERROR / LOADING SHADER") << "Shader has already been loaded." << std::endl;

        exit(-1);

    }

    if(err & gl::ShaderLoadErrorType::VERTEX\_SHADER\_ERROR)

    {

        std::cout << get\_msg("ERROR / LOADING SHADER") << "Compile error occured in vertex shader." << std::endl;

        std::cout << get\_msg("INFO / VERTEX SHADER") << shader.get\_last\_vertex\_msg() << std::endl;

        exit(-1);

    }

    if(err & gl::ShaderLoadErrorType::FRAGMENT\_SHADER\_ERROR)

    {

        std::cout << get\_msg("ERROR / LOADING SHADER") << "Compile error occured in fragment shader." << std::endl;

        std::cout << get\_msg("INFO / FRAGMENT SHADER") << shader.get\_last\_fragment\_msg() << std::endl;

        exit(-1);

    }

    if(err & gl::ShaderLoadErrorType::SHADER\_LINK\_ERROR)

    {

        std::cout << get\_msg("ERROR / LOADING SHADER") << "Link error occured." << std::endl;

        std::cout << get\_msg("INFO / LINK SHADER") << shader.get\_last\_link\_msg() << std::endl;

        exit(-1);

    }

    std::cout << get\_msg("INFO / LOADING SHADER") << "Shader \"" << path + vertex\_fname << "\" and \"" << path + fragment\_fname << "\" loaded." << std::endl;

}

Hier werden die Shader geladen, die für die Grafikszene benötigt werden. Es werden 2 Shader-Stufen verwendet: Vertex-Shader und Fragment-Shader. Der Vertex-Shader wird für jeden Vertex in der Grafikszene aufgerufen und bestimmt die endgültige Position der Vertices im Raum. Anschließend findet OpenGL intern heraus, welche Pixel das aktuelle Dreieck abdeckt und ruft für jeden Pixel den Fragment-Shader auf. Der Fragment-Shader besitzt die Aufgabe die Endgültige Farbe des Pixels zu errechnen. In ihm finden Lichtberechnungen, Schattenberechnungen, usw. statt. Der Vertex-Shader ist in einer .vert und der Fragment-Shader in einer .frag abgespeichert und werden mit der Programmiersprache GLSL programmiert. In diesem Codeabschnitt wird ein Paar aus Vertex- und Fragment-Shader mit der Klassenmethode „shader.load“ geladen. Der Shader Code wird intern kompiliert und gelinkt. Anschließend erfolgen Abfragen, ob beim Kompilieren bzw. Linken Fehler aufgetreten sind.

#### Grafik Renderer mit OpenGL

Zunächst werden Puffer auf im VRAM für die Meshes (3D-Daten) erstellt.

Es gibt verschiedene Arten von Puffer, die verwendet werden:

* Vertex-Puffer: Das sind Puffer, wo die Raumpunkte, Vertices genannt, abgespeichert werden.
* Textur-Puffer: Das sind Puffer, wo die Pixel der Texturen, üblicherweise im 8-Bit-Gleitkommaformat, abgespeichert werden.
* Frame-Puffer: Der Frame-Puffer beinhaltet die gerenderten Farbinformationen der Pixel nach dem Aufruf der Shader.
* Render-Puffer: Dieser Puffer beinhaltet ebenfalls Informationen nach dem Rendern. Jedoch zum Unterschied zum Frame-Puffer beinhaltet der Render-Puffer Tiefen- und Stencil-Informationen. Diese Information wird verwendet, um zu erkennen, ob ein Objekt hinter einem anderen liegt.

Nach dem die Meshes in Puffer geladen wurden, wird das fertige Bild gerendert. Da nicht nur ein statisches Bild gerendert werden soll, muss das Rendering in einer Endlosschleife stattfinden. In dieser Renderschleife werden die Renderbefehle von der CPU an die Grafikkarte gesendet.

In dem folgenden Codeabschnitt findet das Rendering der gesamten 3D-Szene statt:

/\* -----------------------------------------------------------------------------

         \* VISUALIZATION SCENE

         \* -----------------------------------------------------------------------------\*/

        // set flags

        glEnable(GL\_DEPTH\_TEST);

        glEnable(GL\_TEXTURE\_2D);

        /\* GENERATE (DRAW) SHADOW MAP(S) \*/

        glBindFramebuffer(GL\_FRAMEBUFFER, fbo\_worldlight\_shadow);

        glViewport(0, 0, WORLDLIGHT\_SHADOW\_RESOLUTION, WORLDLIGHT\_SHADOW\_RESOLUTION);

        glClear(GL\_DEPTH\_BUFFER\_BIT);

        simple\_depth\_shader.use();

        simple\_depth\_shader.uniform\_matrix\_4x4f("lightSpaceMat", 1, false, glm::value\_ptr(light\_space\_transform));

        // draw table to depth map

        glBindVertexArray(vao\_table);

        glDrawArrays(GL\_QUADS, table\_mtl.vertex\_offset, table\_mtl.vertex\_count);

        glBindVertexArray(0);

        // draw vehicles to shadow map

        glBindVertexArray(vehicle\_buffer.get\_vao());

        glDrawArraysInstanced(GL\_QUADS, vehicle\_mtl.vertex\_offset, vehicle\_mtl.vertex\_count, vehicle\_buffer.get\_num\_vehicles());

        glBindVertexArray(0);

        /\* DRAW OBJECTS \*/

        glBindFramebuffer(GL\_FRAMEBUFFER, fbo\_vis);

        glViewport(0, 0, vis\_window\_handler.get\_width(), vis\_window\_handler.get\_height());

        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

        glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

        // set flags

        glEnable(GL\_BLEND);

        glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

        glEnable(GL\_CULL\_FACE);

        // set light values for used shaders

        vis\_shader.use();

        set\_light\_values(vis\_shader);

        // set view-projection matrix

        vis\_shader.uniform\_matrix\_4x4f("VP", 1, false, glm::value\_ptr(glm::mat4(vis\_projection \* vis\_view)));

        vis\_shader.uniform\_matrix\_4x4f("lightSpaceMat", 1, false, glm::value\_ptr(light\_space\_transform));

        /\* DRAW TABLE \*/

        draw\_table(vis\_shader, table\_mtl, vao\_table, table\_diffuse\_map, table\_specular\_map, shadow\_maps);

        /\* DRAW VEHICLES \*/

        draw\_vehicles(vis\_shader, vehicle\_mtl, vehicle\_buffer.get\_vao(), vehicle\_buffer.get\_num\_vehicles(), vehicle\_diffuse\_map, vehicle\_specular\_map, shadow\_maps);

       /\* -----------------------------------------------------------------------------

        \* SCENE (GUI + VISUALIZATION)

        \*

        \* Everything gets rendered to a seperate framebuffer.

        \* Then the output is rendered as a single texture to a quad to be able to

        \* make post-processing effects with the scene (e.g. gamma-correction).

        \* -----------------------------------------------------------------------------\*/

        glBindFramebuffer(GL\_FRAMEBUFFER, fbo\_scene);

        glViewport(0, 0, main\_window\_handler.get\_width(), main\_window\_handler.get\_height());

        glClearColor(Main::BACKGROUND\_COLOR[0], Main::BACKGROUND\_COLOR[1], Main::BACKGROUND\_COLOR[2], Main::BACKGROUND\_COLOR[3]);

        glClear(GL\_COLOR\_BUFFER\_BIT);

        // set flags

        glDisable(GL\_DEPTH\_TEST);

        glDisable(GL\_CULL\_FACE);

        /\* DRAW GUI \*/

        draw\_GUI(element\_shader, text\_shader, element\_renderer, element\_font);

        /\* DRAW VISUALIZATION SCENE \*/

        draw\_fb(fb\_scene\_shader, vao\_fb, tex\_vis, 0);

        /\* -----------------------------------------------------------------------------

         \* SCREEN (SCENE)

         \* -----------------------------------------------------------------------------\*/

        glBindFramebuffer(GL\_FRAMEBUFFER, 0);

        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

        glClear(GL\_COLOR\_BUFFER\_BIT);

        // set flags

        glDisable(GL\_BLEND);

        /\* DRAW SCENE \*/

        draw\_scene(scene\_shader, vao\_fb, tex\_scene, 4);

        // disable remaining active flags

        glDisable(GL\_TEXTURE\_2D);

        glUseProgram(0);    // unuse shader

        glfwSwapBuffers();

        main\_running = !glfwGetKey(GLFW\_KEY\_ESC) && glfwGetWindowParam(GLFW\_OPENED);

        glEndQuery(GL\_TIME\_ELAPSED);

        int time;

        glGetQueryObjectiv(timer\_query, GL\_QUERY\_RESULT, &time);

        time\_results.push\_back(time);

Zunächst wird eine Sogenannte “Shadow Map“ gerendert. Eine Shadow Map enthält nur Tiefeninformation und wird verwendet, um Schatten zu rendern. Dazu wird der gesamte Inhalt der 3D-Szene aus Sicht der Lichtquelle gerendert, um danach entscheiden zu können, ob ein Pixel im Schatten oder im Licht ist.

Danach wird die eigentliche 3D-Szene mit Farbinhalt und allem was dazugehört gerendert. Dar Renderinhalt wird in einen separaten Frame-Puffer gerendert, sodass zum Schluss noch Pixel-Post-Processing Operationen durchgeführt werden können.

Zuletzt wird das GUI zur Szene dazugerendert und es werden Post-Processing Operationen, wie Gamma-Korrektur durchgeführt.

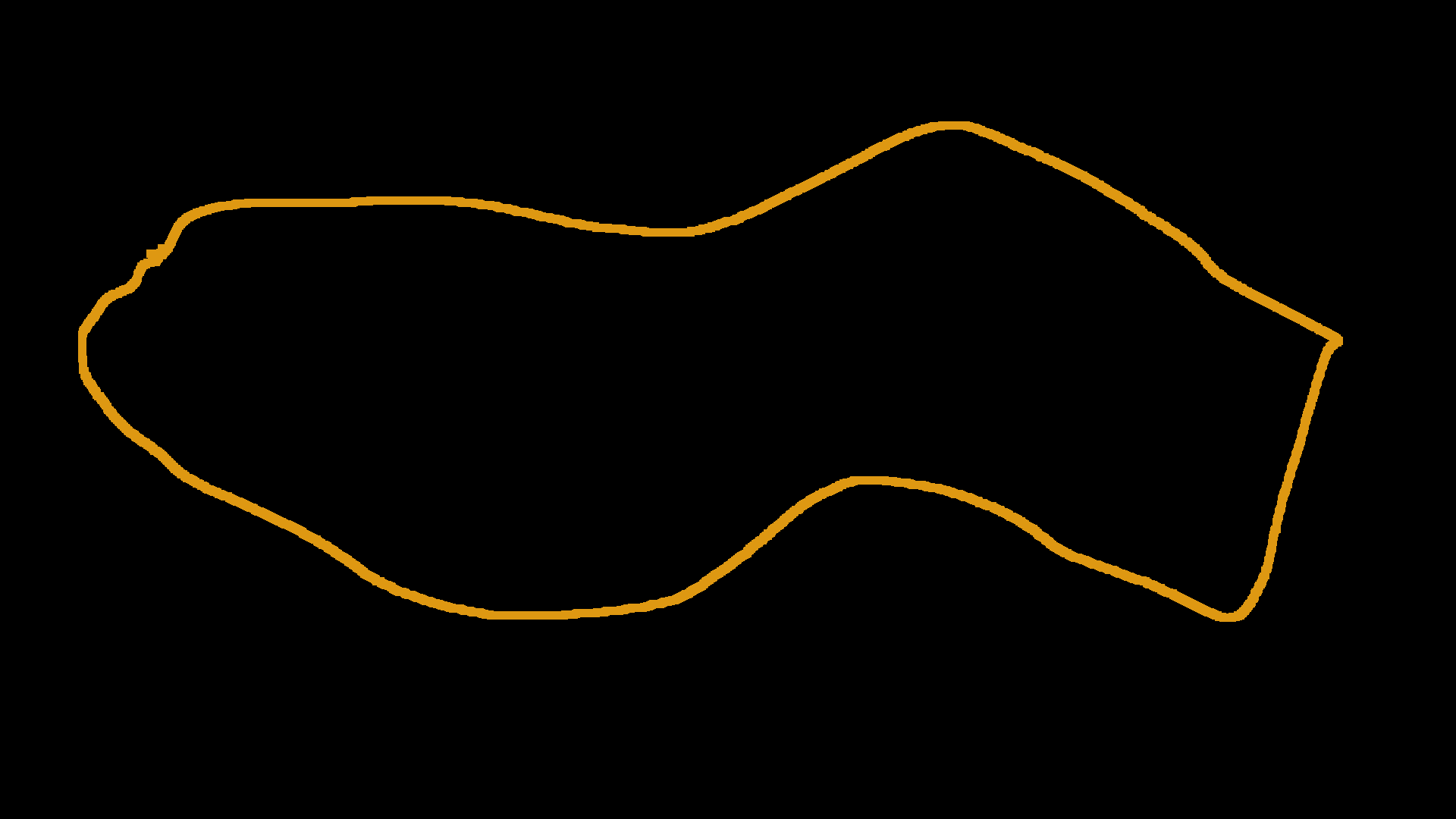
### Fahrzeug AI

#### Generierung des Weges

Der Weg wird aus einer Bilddatei generiert.   
Dieser Algorithmus ist in 3 Teile aufgeteilt:

1. Das Bild zu einem Graustufenbild zu konvertieren.
2. Die Außenlinie des Weges finden.
3. Die Außenlinie in eine bestimmte Anzahl an Punkten unterteilen.

**Originalbild:**



**Konvertierung zu einem Graustufenbild:**

void to\_grayscale(uint8\_t\*\* data, image\_info\_t& ii)

{

    /\* Grayscale: 1 color-channel that means a multiplication with the number of channels is useless.

    \*                                               |

    \*                                               \/  here

    \*/

    uint8\_t\* img\_gray\_data = new uint8\_t[ii.width \* ii.height]; // Allocating the new image buffer for the grayscale image.

    image\_info\_t gray\_ii{ii.width, ii.height, 1};               // And also declare a new image\_info\_t struct for it.

    // 2-dimensional loop that iterates every pixel of the image.

    int x, y;   // a optimization for speed

    for(y = 0; y < ii.height; y++)

    {

        for(x = 0; x < ii.width; x++)

        {

            unsigned int sum = 0;

            // Calculate the average value of all color components except the alpha channel if the image has one.

            // The average value of the 3 colors (RGB) corresponds to the grayscale value.

            for(int c = 0; c < ((ii.channels < 4) ? ii.channels : 3); c++)

            {

                sum += (\*data)[img\_at(x, y, ii) + c];

            }

            // Write the grayscale value to the new generated buffer.

            img\_gray\_data[img\_at(x, y, gray\_ii)] = sum / ii.channels;

        }

    }

    ii = gray\_ii;               // update the image information

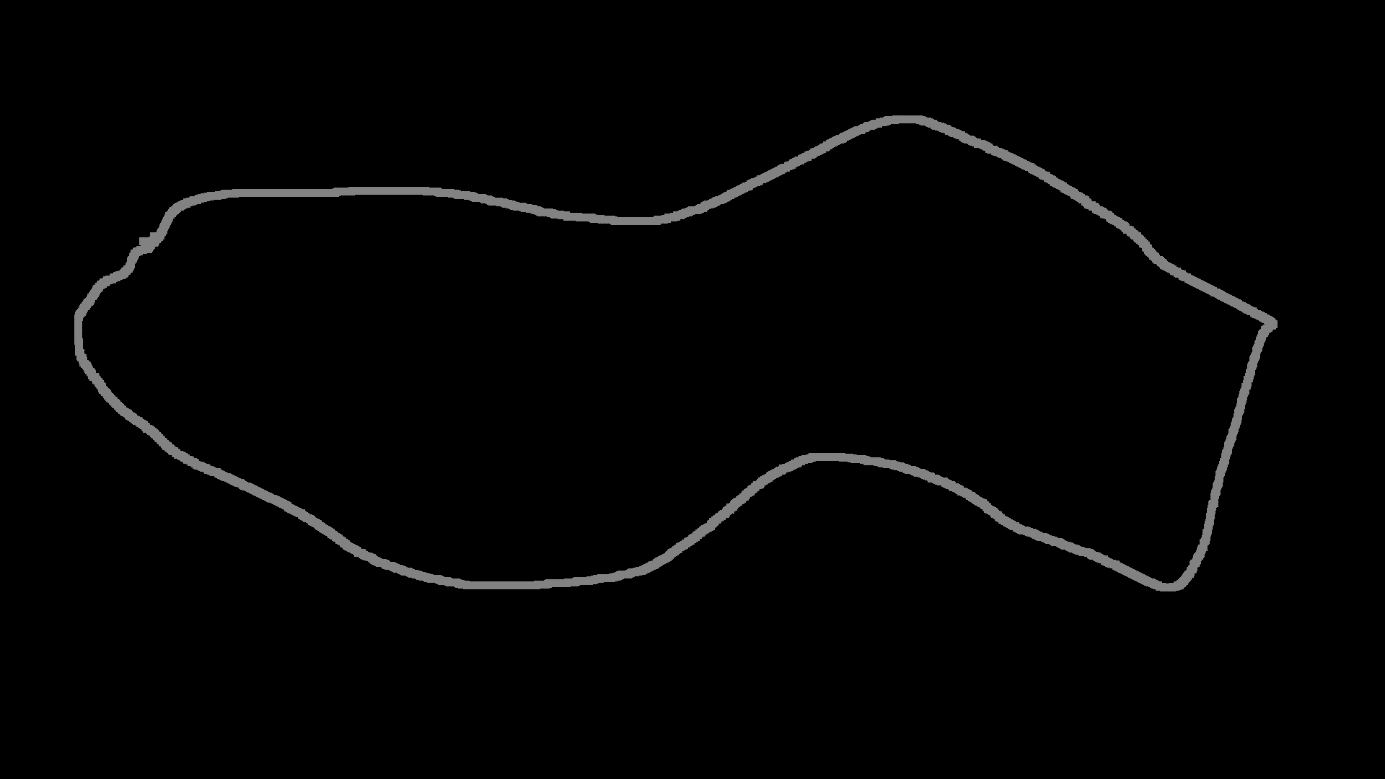
    stbi\_image\_free(\*data);     // free the old data

    \*data = img\_gray\_data;      // let the data-pointer point to the new data (grayscale image data)

}

Um ein RGB Bild zu einem Graustufenbild zu konvertieren wird folgende Formel verwendet:

also es wird der arithmetische Mittelwert aus den 3 Farbkomponenten gebildet. Enthält die Bilddatei jedoch noch eine 4. Komponente, zum Beispiel einen Alpha-Anteil, wird diese einfach weggelassen.   
Das Bild sieht danach so aus:



**Außenlinie des Wegs finden:**

Um eine Außenlinie bestimmen zu können muss zunächst einmal klar sein, was ist überhaupt Weg und was nicht. Um das bestimmen zu können benötigt man einen Schwellenwert. Dieser Schwellenwert ist sehr niedrig angesetzt. Der Graustufenwert 0 bedeutet komplett schwarz und 255 bedeutet komplett weiß. Der Schwellenwert hat einen Wert von 10:

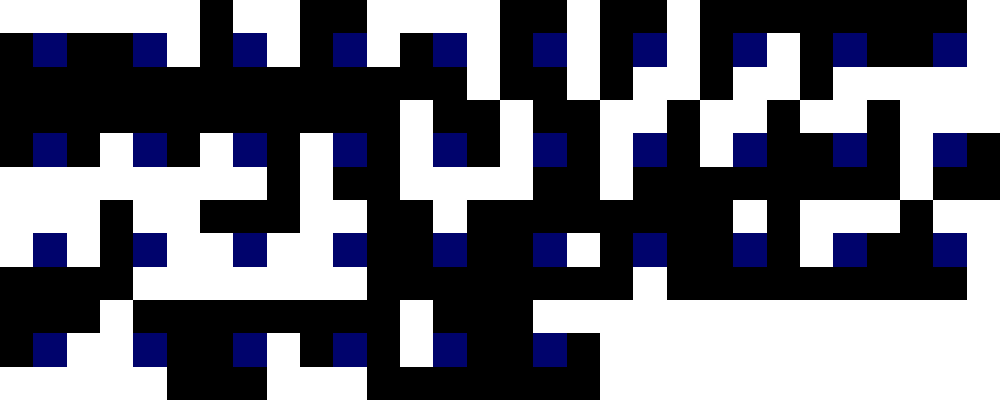
#define PATH\_THRESHOLD 10

Das heißt, dass jeder Farbwert unter 10 als Hintergrund erkannt wird und alles größer oder gleich 10 als Weg erkannt wird.

Um die Außenlinie zu finden wird zunächst das Bild durchgegangen und nach einem Anfang gesucht. Nach dem der Anfang gefunden wurde, wird der Weg im Urzeigersinn nachgefahren und jeder Pixel vom Rand abgespeichert.

Um das nächste Pixel des Randes finden zu können, muss bekannt sein wie die Umgebung, des aktuellen Pixels aussieht. Dazu wird eine 3x3 Matrix, um das Aktuelle Pixel eingelesen und die Richtung bestimmt, in der sich das nächste Randpixel befindet.

Hier sieht man alle Möglichkeiten, die es gibt, um die Umgebung des aktuellen Pixels eindeutig zu bestimmen (das aktuelle Centerpixel ist blau markiert, schwarz sind die Umgebungspixel, die zum Weg dazugehören und weiß sind die Umgebungspixel, die Hintergrund sind):



Anmerkung: Um den Weg mit einer 3x3 Matrix eindeutig bestimmen zu können, muss der Weg mindestens 3 Pixel breit sein, sonst würden sich undefinierte zustände ergeben und der Weg kann nicht generiert werden.

Hier ist der Code des beschrieben Algorithmus, um die Außenlinie zu finden:

void gen\_path(uint8\_t\* data, std::vector<img\_coord\_t>& path, const std::map<Path::PathMatrix\*, Path::PathDirection>& m2d, const image\_info\_t& ii)

{

    // If the image has no pixels, there is no need to generate a path.

    if(ii.width == 0 || ii.height == 0)

        return;

    int bx = 0, by = 0;                         // Not only a optimization for speed, the variables are also used later in the code!!!

    Path::PathMatrix begin\_mat;                 // Matrix of the begin pixel.

    const Path::PathDirection\* dir = nullptr;   // Direction of the matrix initialized to 'nullptr'.

    // Iterate through the pixels of the image.

    // The loop breaks if the image was iterated through if no path was found or, which is more likely,

    // the loop breaks at the first pixel that has a valid matrix.

    for(by = 0; by < ii.height && dir == nullptr; by++)

    {

        for(bx = 0; bx < ii.width && dir == nullptr; bx++)

        {

            begin\_mat = gen\_pathmatrix(data, bx, by, ii);   // Generate matrix of current pixel.

            dir = get\_dirp(m2d, begin\_mat);                 // Get the direction.

        }

    }

    // Subtract 1 because the loop iterates through one time too much.

    int x = --bx;

    int y = --by;

    // Current matrix.

    // Initialize it to the begin matrix because this is the starting point.

    Path::PathMatrix cur\_mat = begin\_mat;

    do

    {

        // Get the direction of the current matrix.

        dir = get\_dirp(m2d, cur\_mat);

        // Maybe the direction is invalid which means the pointer points to 'nullptr'.

        if(dir != nullptr)

        {

            path.push\_back({x, y});     // Push the current x and y value into the vector because this is a valid path-coordinate.

            x += dir->direction\_x();    // Add the direction to the x and y value which is returned by the direction\_x() and direction\_y() method.

            y += dir->direction\_y();

        }

        cur\_mat = gen\_pathmatrix(data, x, y, ii);   // Generate the next matrix out of the new x and y value.

    }

    while(!(x == bx && y == by) && dir != nullptr); // Break if the begin matrix has been reached again or if the direction has become invalid.

}

Hier ist ein herangezoomter Ausschnitt, wo die Randlinie (weiß gekennzeichnet) gut zu sehen ist:



**Außenlinie in einzelne Punkte (sog. Goals) unterteilen:**

Die Anzahl der Pixel der Außenlinie wird durch die Anzahl der Punkte, die generiert werden sollen, unterteilt. Somit herhält man wie viele Randpixel von Punkt zu Punkt sind.

Die resultierenden Punkte werden zum Schluss in einem File abgespeichert.

Dazu der Codeausschnitt, um die Randpixel zu unterteilen:

bool gen\_goals(const std::vector<img\_coord\_t>& path, std::vector<goal\_coord\_t>& goals, unsigned int num\_goals)

{

    // Invalid number of goals:

    //  If the number of goals is to big or if 'path' does not contain any coordinates.

    if(num\_goals > path.size() || path.size() == 0)

        return false;

    // Calculate the width between the goals.

    float g2g\_width = (float)path.size() / (float)num\_goals;

    // Split the path into a certain number of goals.

    for(float i = 0; i < path.size(); i += g2g\_width)

    {

        goals.push\_back(path.at((size\_t)i));

    }

    // Because the vehicle should drive back to the begin whre it started,

    // the first goal has to be pushed at the end of the goal-vector.

    // Only if the vector contains at least one element.

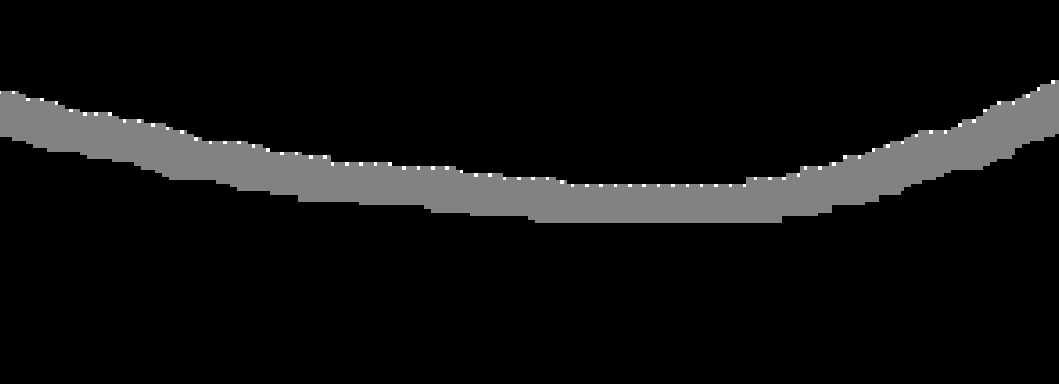
    if(goals.size() > 0)

        goals.push\_back(goals.at(0));

    return true;

}

So sieht das in dem Endresultat im Beispielbild aus, auch hier wieder ein herangezoomter Ausschnitt:



Dieser Beispielweg wurde in 1000 Punkte unterteilt.

#### Aufbereitung der Goals

Jedes Fahrzeug besitzt eine Numerische ID beginnend bei 0. Die Goals werden von dem File, welches vom Pathgenerator erstellt wurde, gelesen und zu dem entsprechenden Fahrzeug zugeordnet. Wurde dem Fahrzeug bereits ein Weg zugeordnet wird der alte Überschrieben.

Dazu der Codeausschnitt:

/\*  If the path has successfully been generated, the goals can be

                    \*   read from the file.

                    \*/

                    FILE\* file = fopen("pathgenerator/goals.gol", "r");   // Open the file, the filename is defined.

                    if(file == NULL)

                    {

                        /\*  If the file could not be opened the output file could not be opened

                        \*   or the file has been deleted.

                        \*/

                        gettime(time);

                        fprintf(shared\_variables.logfile, "[%s] [ERROR] Could not find path to goal-output file.\n", time);

                        send\_error(&server.get\_socket(0), Schwarm::packet\_error::PACKET\_FAILED\_GENERATING\_PATH);

                    }

                    else

                    {

                        /\* If the file can be open, the reading process can begin. \*/

                        // Clear the goal vector to write new goals into it.

                        goals[shared\_variables.pathgenpacket.get\_vehicle\_id()].clear();

                        Goal goal;

                        gettime(time);

                        fprintf(shared\_variables.logfile, "[%s] [INFO] Reading goals...\n", time);

                        while(!feof(file))  // Read the whole file...

                        {

                            fscanf(file, "%f %f\n", &goal.x, &goal.y);          // Read one line and put the values into the "Goal" struct.

                            goals[shared\_variables.pathgenpacket.get\_vehicle\_id()].push\_back(goal); // Push the goal to the goal vector.

                        }

                        gettime(time);

                        fprintf(shared\_variables.logfile, "[%s] [INFO] Successfully generated goals for vehicle %d.\n", time, shared\_variables.pathgenpacket.get\_vehicle\_id());

                        fclose(file);   // Close the previously opened file.

                        // Send acnoledge.

                        Schwarm::AcnPacket packet;

                        packet.allocate(packet.min\_size());

                        packet.encode();

                        server.get\_socket(0).send(packet.rawdata(), packet.size(), 0);

#### Senden der Goals zur Visualisierung

Die ganze Kommunikation der Aufbereitung der Goals (Path-Server) mit der Visualisierung findet über TCP statt.

Zunächst teilt die Visualisierung mit dem im Punkt 3.2.2.1 genannten Befehlt „/simu generate“ dem Path-Server mit, dass ein neuer Weg für eine im Befehl mitgegebene Fahrzeugnummer generiert werden soll. Die generierten Goals werden somit zum angegebenen Fahrzeug abgespeichert.

Danach kann die Visualisierung die einzelnen Goals empfangen. Dies geschieht über eine Anfrage an den Path-Server. Bei dieser Anfrage muss mitgegeben werden, für welche Fahrzeugnummer wird das Goal gesendet wird und das wievielte Goal es sein soll. Wird jedoch ein Goal angefragt, den es nicht gibt, wird ein Error an die Visualisierung zurückgesendet. Hingegen gibt es das Goal, wird ein Paket gesendet welches die Fahrzeug ID, die X-Pixelkoordinate und die Y-Pixelkoordinate beinhaltet.

Dazu der Codeausschnitt:

void process\_packet(cppsock::socket\* socket, uint8\_t\* data, void\*\* persistant)

{

    const uint8\_t\* id = Schwarm::Packet::id\_ptr(data);              // Get pointer to packet id.

    const size\_t\* size = Schwarm::Packet::size\_ptr(data);           // Get pointer to size of packet.

    SharedVariables\* shared\_variables = (SharedVariables\*)\*persistant;    // Get pointer to shared memory.

    char time[48];

    gettime(time);

    if(\*id == Schwarm::ExitPacket::PACKET\_ID)

    {

        /\*  If exit command was received set running value to 'false'.

        The server will shut down. \*/

        fprintf(shared\_variables->logfile, "[%s] [INFO] Received stop command.\n", time);

        shared\_variables->running = false;

    }

    else if(\*id == Schwarm::PathGeneratePacket::PACKET\_ID)

    {

        /\*  Only process the packet if the main thread is still running (highest priority)

            and only if the server is not already generating a path. \*/

        if(shared\_variables->running && !shared\_variables->generating\_path)

        {

            shared\_variables->pathgenpacket.allocate(\*size); // Allocate memory for the packet.

            shared\_variables->pathgenpacket.set(data);       // Set the data string.

            shared\_variables->pathgenpacket.decode();        // Decode the packet.

            fprintf(shared\_variables->logfile,

                    "[%s] [INFO] Generating goals for file %s with %u goals for vehicle %d...\n",

                    time,

                    shared\_variables->pathgenpacket.get\_filepath(),

                    shared\_variables->pathgenpacket.get\_num\_goals(),

                    shared\_variables->pathgenpacket.get\_vehicle\_id());

            // Set values for shared memory.

            shared\_variables->generating\_path = true;     // Set "generating\_path" to 'true' to let the rest of the server know that a path is being generated.

            shared\_variables->packet\_id = shared\_variables->pathgenpacket.id();

            /\*  NOTE: It is important that setting the packed id is the last operation to ensure a synchronization

            \*   betwenn this and the main thread. The main thread will only start to process the packet if the

            \*   id has been set.

            \*/

        }

        else if(shared\_variables->running && shared\_variables->generating\_path)

        {

            // If the server is busy generating goals, send an error to the client.

            fprintf(shared\_variables->logfile, "[%s] [INFO] Server is already generating goals.\n", time);

            send\_error(socket, Schwarm::packet\_error::PACKET\_SERVER\_BUSY);

        }

    }

    else if(\*id == Schwarm::GoalReqPacket::PACKET\_ID)

    {

        fprintf(shared\_variables->logfile, "[%s] [INFO] Received goal request.\n", time);

        if(shared\_variables->running && !shared\_variables->generating\_path)

        {

            shared\_variables->goalreqpacket.allocate(\*size); // Allocate memory for the packet.

            shared\_variables->goalreqpacket.set(data);       // Set the packet data.

            shared\_variables->goalreqpacket.decode();        // Decode the packet.

            /\*  The sending mechanic takes place in the main thread because

            \*   the vector for the goals is located threre.

            \*/

            // Set values for shared memory.

            shared\_variables->packet\_id = shared\_variables->goalreqpacket.id();    // The same thing with the id like before...

        }

        else if(shared\_variables->running && shared\_variables->generating\_path)

        {

            // If the server is busy generating goals, send an error to the client.

            fprintf(shared\_variables->logfile, "[%s] [INFO] Can't send goal because server has not finished generating goals\n", time);

            send\_error(socket, Schwarm::packet\_error::PACKET\_SERVER\_BUSY);

        }

    }

}

#### Verarbeitung der Goals in der Visualisierung

Befindet sich die Visualisierung im Simulationsmodus werden nur Daten vom Path-Server empfangen, verarbeitet und am Bildschirm dargestellt.

Befindet sich die Visualisierung nicht im Simulationsmodus, also steuert sie den realen Aufbau an, werden zusätzlich noch die Positionen der realen Fahrzeuge, gesendet von der Positionserkennungs-Software, empfangen. Diese Positionen werden ebenfalls verarbeitet, nämlich es wird sich eine Abweichung daraus berechnet, die die aktuelle Position und Richtung der Fahrzeuge korrigiert. Darüber hinaus werden sich befehle errechnet, wie „links fahren, „rechts fahren“, etc., die verwendet werden, um die Fahrzeug-Software anzusteuern.

Genauer wird auf den Auswertealgorithmus im Punkt 3.6 eingegangen.

## Erstellung von 3D-Modellen

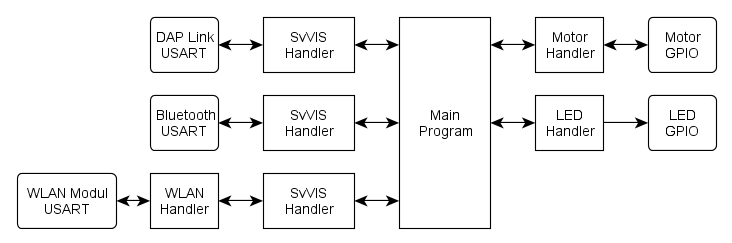
## Bewegungen im Dreidimensionalen Raum

## Auswertung von erhaltenen Positionsdaten

## Übertragung zu Swarm Controll

# Fahrzeug Software

## Übersicht der Architektur



## Kommunikation mit Swarm Controll

Ein Fahrzeug kann über 3 verschiedene Wege mit der Außenwelt kommunizieren:

1. Serielle Schnittstelle über DAP-Link Adapter
2. Serielle Schnittstelle über Bluetooth mittels HC06 Modul
3. TCP/IP Kommunikation mittels ESP8266 Modul

Da die Kommunikation über den DAP-Link Adapter nur für Entwicklungs- und Testzwecke verwendet werden kann, wird die Kommunikation im Endaufbau über Bluetooth oder WLAN stattfinden.

### Kommunikationsprotokoll

Die Software des Fahrzeugs kommuniziert über das SvVis Protokoll mit der Swarm Control. Dieses Protokoll besteht aus einem Header mit der Größe von 1 Byte und darauffolgenden Nutzdaten. Die länge der Nutzdaten ist vom Header und den Daten selbst abhängig.

|  |  |  |
| --- | --- | --- |
| Header Byte(s) | Bedeutung | Länge der Daten |
| 0x0A  10 | String | Daten werden mit '\0' beendet. Implementierung ist auf eine String-Länge von 31 begrenzt |
| 0x0B-0x13  11-19 | 16-bit integer (int16\_t) | 2 Bytes  16 Bit |
| 0x15-0x1D  21-29 | 32-bit floating point number (float) | 4 Bytes  32 Bit |

## Hardware Ansteuerung

### LED-Ansteuerung

Die Ansteuerung der LEDs ist die simpelste Hardware-Ansteuerung des Fahrzeuges. Es gibt eine Led, die periodisch Blinkt, um eine visuelle Bestätigung zu geben, dass die Software des Autos aktiv ist. Außerdem kann die Software die 3 Positionserkennungsleds einzel ein- und ausschalten.

### Kommunikationsmodule

#### DAP Link

Die serielle Schnittstelle über den DAP\_Link ist hauptsächlich für die Softwareentwickling verwendbar, da ein USB-Kabel für die Kommunikation benötigt wird.

#### Bluetooth

Das verwendete HC06 Modul erlaubt kabellose Datenübertragung mit der Fahrzeugsoftware. Allerdings ist die Reichweite dieser Verbindung nicht sehr groß

#### WLAN

Das ESP8266 Modul ermöglicht die Kommunikation über eine TCP-Verbindung, die die TCP-Pakete über WLAN sendet und empfängt.

Allerdings müssen dazu einige Details über die zu verwendende Verbindung in die Konfigurationsdatei des Programms Eingetragen werden:

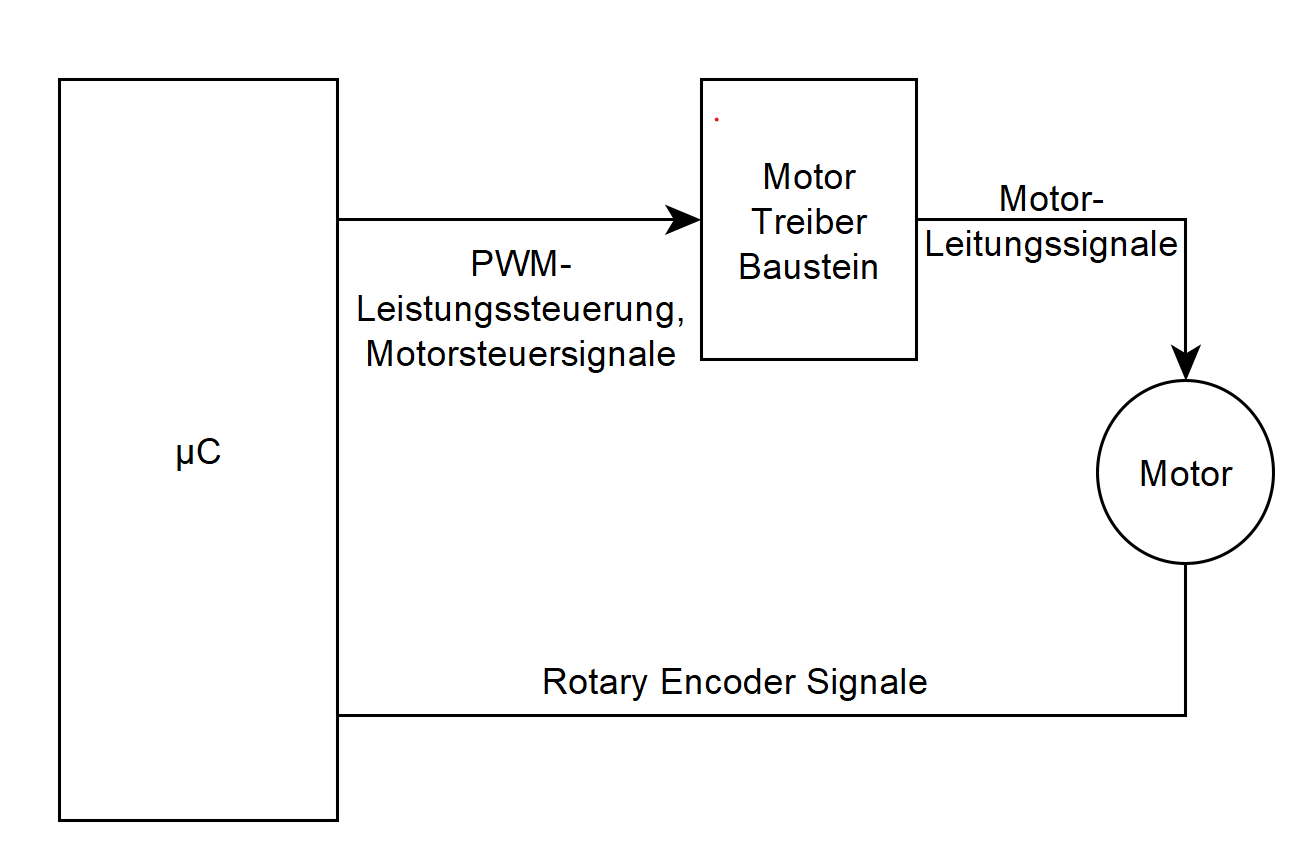
* SSID des WLAN Access Points
* Password für das WLAN-Netzwerk
* IP Adresse der Swarm Control
* TCP Port der Swarm Control

Die Ansteuerung des WLAN Moduls erfolgt über AT-Kommandos über eine Serielle Schnittstelle.

### Motor Ansteuerung

Die Ansteuerung der Motoren ist einer der wichtigsten Komponenten der Auto-Software, da dieser Teil dafür verantwortlich ist, dass sich aus Auto bewegt.

Die Motoren werden mithilfe eines Motortreiberbausteins angetrieben. Dieser Motortreiberbaustein bietet die Möglichkeit, die Motorleistung mit einem PWM Signal zu regeln. Der Verwendete Mikrocontroller (STM32F107RB) bietet eine Möglichkeit, ein PWM-Signal mit einstellbaren duty-cycle über die Integrierten Timer-Peripherie Komponenten generieren zu lassen. Die Motoren haben zusätzlich Inkrementalgeber, um die derzeitige Drehzahl des Motors auslesen zu können.



Eine Verbesserung dieses Designs wäre ein Hardwaremäßiger Rotary Decoder Baustein, der die Decodersignale auswertet und dadurch Arbeit des Microcontrollers abnimmt.

Eine weiter Möglichkeit wäre Schrittmotoren zu verwenden, da Schrittmotoren keine Drehzahlregelung mit Rückmeldung benötigen. Außerdem Kann mit Schrittmotoren eine wesentlich genauere Positionierung des Fahrzeugs erreicht werden.

# Fahrzeug Software

## Übersicht der Architektur

## Kommunikation mit Swarm Controll

### Kommunikationsprotokoll

## Hardware Ansteuerung

### LED-Ansteuerung

### Kommunikationsmodule

### Bluetooth

### WLAN

### Motor Ansteuerung

# Ergebnisse

## Funktionalität Positionserkennung

## Steuerung der Fahrzeuge

## Simulationstest mit Pseudodaten

## Steuersoftwarte Funktionalitätstest

## Schwarmbewegung

### Kreis

### Zick-Zack

# Anhang

## Inbetriebnahme (f. 4Klasse TdoT)

## Projektmanagement

## Projektplan

## Projekttagebuch

## Projektkosten

# Quellenverzeichnis

## Bücher

## Onlinemedien

## Zeitschriften