AUTONOMOUS LINE-FOLLOWING CAR

MIR SHARJIL HASAN
Electronic Engineering
Hamm-Lippstadt
Lippstadt, Germany
mir-sharjil.hasan@stud.hshl.de

TALHA KHAN
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany

MUHAMMAD HAMAS
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany

June 2025

Abstract

This project undertakes the design and development of an intelligent line follower vehicle that can detect obstacles and respond to them in an intelligent manner. The chassis of the vehicle was designed using SolidWorks and Fusion 360, modelled with laser cut parts and 3D printed components. Line sensors are used for the predefined path, while ultrasonic sensors detect obstacles around the vehicle for dynamic avoidance. Colour sensors enable the detection of the colours of obstacles which then trigger specific behaviours by the vehicle. These different types of sensing give the vehicle smartness in decision-making and adaptability to its environment. Therefore, a robust, reliable, autonomous navigation system able to work in dynamic environments is achieved.

Key Topics: SysML, Line-Following Robot, PLA, Detecting Obstacles, Merging Sensor Data **Index Terms:** SysML, Line-Following Robot, PLA, Obstacle Detection, Sensor Fusion

1 Introduction

The goal of this project was to design and build a line-following bot that can not just follow a marked path but also see obstacles and act on their colour. This joins traditional way-following methods with real-time object spotting and smart choice-making. The work shown here uses ideas from embedded systems, sensor joining, and mechanical design to make a real self-driving car. The report is organised as follows: Section 2 discusses how SysML was used to model and analyze the system. Section 3 walks through the design and development process. Section 4 outlines the electronic components and explains their roles.

2 System Analysis with SysML

In order to properly scope and manage the project, we adopted SysML (Systems Modelling Language). This allowed us to maintain support throughout the entire development process—from system requirements definition all the way through design and testing. We captured desired functions of the robot using requirement diagrams, and how interactions between different components happen over time with sequence diagrams. These two artefacts have been critical in making sure all subsystems are both aligned and working cohesively [1].

2.1 Requirements

To guide the design and development of our autonomous vehicle prototype, we used a step-by-step requirement diagram, as shown in Figure 1. The diagram outlines all the significant functional requirements the system must fulfil, which serves to offer clear and consistent division of project objectives. At the top level, the general goal is to design an autonomous vehicle (ID=001) to independently drive a track without any human interaction. This is the overall requirement broken down into three primary functional areas:

1. **Track Management (ID=002)**: This requirement blames the vehicle for keeping up with the track and reacting to change, such as curves or sharp turns. It was complemented by a derived requirement, titled Speed Optimisation (ID=006). It asks the vehicle to be able to accelerate or decelerate based on track complexity to make stable and optimal motion [1].

2. **Obstacles Management (ID=003)**: The vehicle should be able to perceive the obstacle ahead and make decisions about it. There are two additional layers in this section:

   - **Braking Distance (ID=007)**: The system must begin responding when it perceives an object at least 3 cm in front [1].
   - **Colour Management (ID=005)**: The system must perceive the color of the obstacle—red, green, or blue—and respond differently in turn. Test case ("Test Colour") was used to test this functionality while developing the application [1].

3. **Direction Management (ID=004)**: This is the turn and manoeuvring criterion while driving on various paths, including parking. Out of this, we drew out a more specific goal:

    - **Drive Different Track Routings (ID=008)**: This means that the car must be capable enough to drive along complex paths and park when required. It is also divided into three specific routing capabilities:
        - Turn 90 Degrees (ID=009)
        - Drive in an Oval (ID=010)
        - Parking (ID=011)

Collectively, these specifications guarantee the system is able to navigate lines accurately, deal with obstacles smartly, and accomplish various navigation tasks. The diagram was used as a basis for both design and testing, making it possible to construct the prototype incrementally while verifying all essential behaviours were included [1].

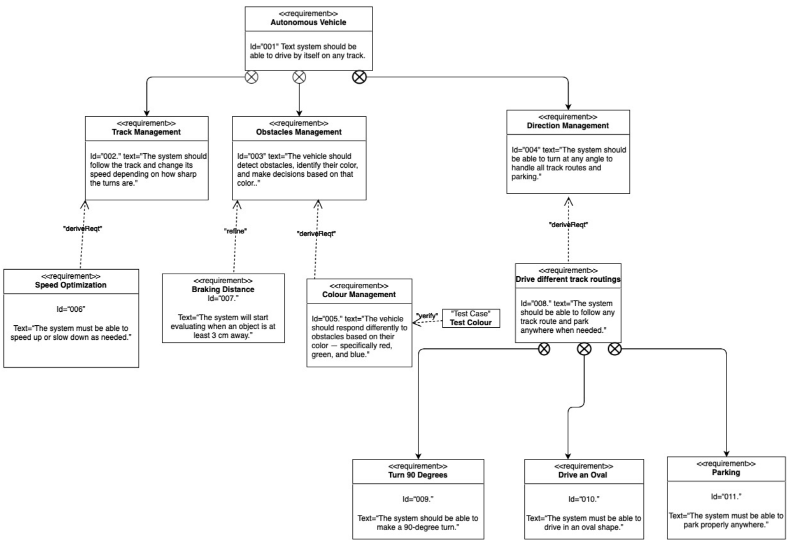

Figure 1: SysML Requirements Diagram for the Autonomous Vehicle

2.2 State Machine Diagram

The state machine diagram represents the behaviour of an autonomous vehicle starting from the Power On state. It first enters the Initialisation state, where sensors and motors are activated, the line sensor is calibrated, and the battery is checked. Once initialisation is complete, the vehicle transitions to the Line Following state, where it continuously follows a line on the track. If an obstacle is detected within 10 cm, it enters the ObstacleDetection state, stopping the motors and using sensors to detect the obstacle and its colour. If no path is available, the vehicle moves to a Stop state. If an alternate path is available, it transitions to Calculate AlternatePath, scans the environment, computes a new route, and rotates to the new path. Once a valid path is found, it resumes line following [1].

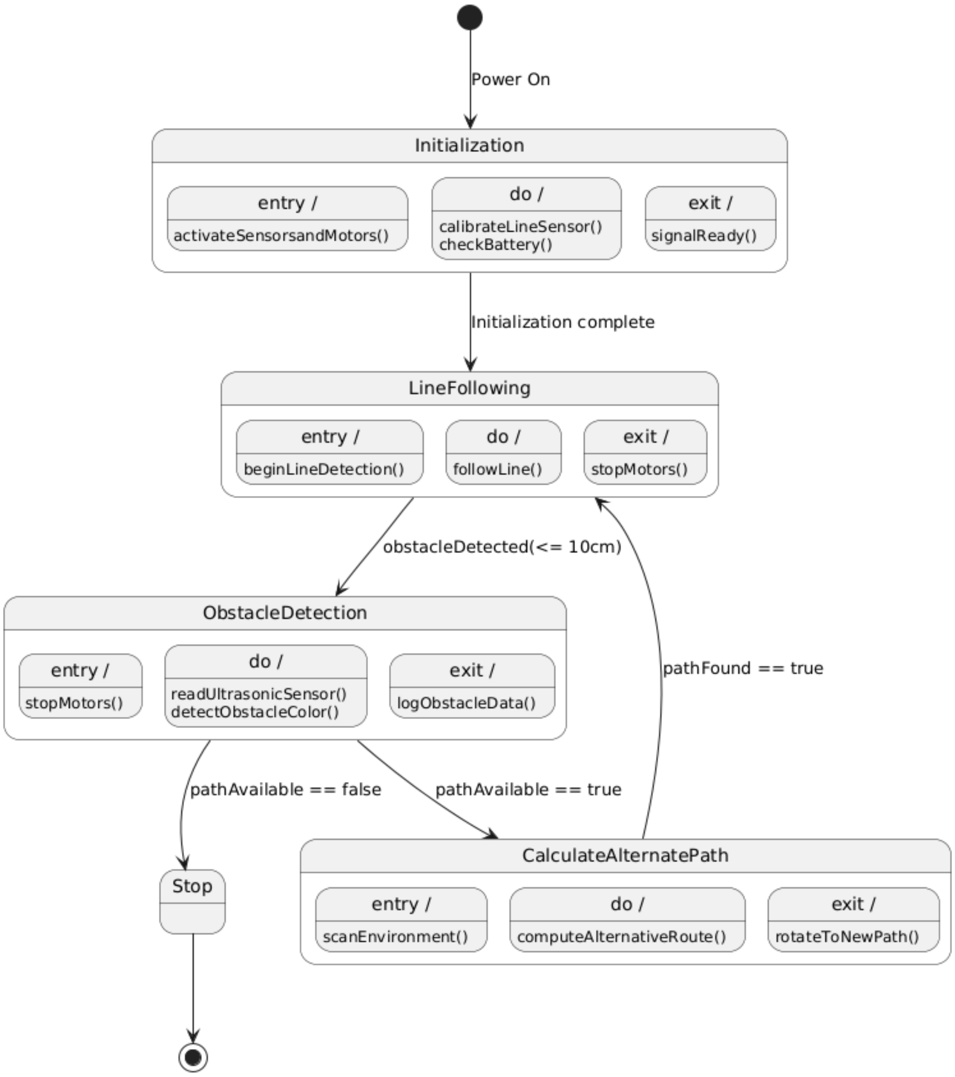

Figure 2: SysML State Machine Diagram of the Autonomous Vehicle

2.3 Activity Diagram

The Activity Diagram depicts the behaviour of the autonomous vehicle from a power on state to an avoiding obstacle state. Upon turning on the system the vehicle's behaviour starts with the detection of the line. The vehicle is programmed to detect curves or obstacles. When this occurs, the system determines which direction to turn. If a curve is detected, then the system will activate either the left motor or right motor in reverse to be able to adjust. If the vehicle does not detect a change in the line, it will continue to accelerate moving perpendicularly to the right line while moving forward. While in motion, the system actively checks for obstacle detection. When an object is detected, the vehicle runs the avoid routine and loops back to following the line. The Activity Diagram shows control flow pertinent to decision making and navigation logic [1].

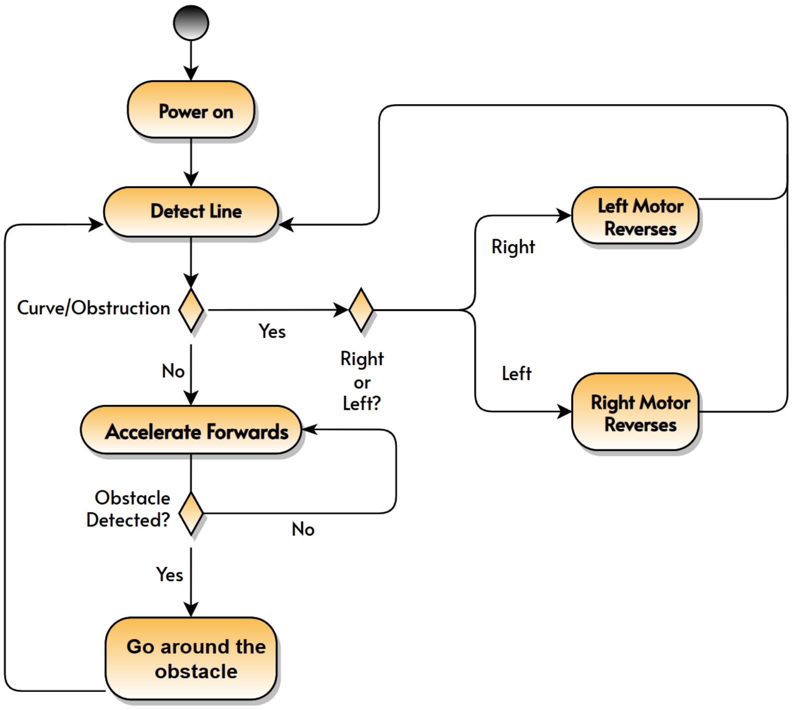

Figure 3: SysML Activity Diagram of the Autonomous Vehicle

2.4 Internal Block Diagram Overview

Internal Block Diagram is the physical structure of the autonomous vehicle's subsystems and how they interact. The four main units of the system are Power System, Sensor System, Controller Unit, and Motor System.

- **Power System**: The battery supplies power to all the main components via separate PowerPorts, ensuring a continuous power supply to sensors, controller, and motors [1].
- **Sensor System**: Includes an Ultrasonic Sensor, IR Sensor, and Colour Sensor. These are all powered and send data to the Controller Unit through DataPorts to enable detection of the environment and identification of obstacles [1].
- **Controller Unit**: Acts as the primary processor. It takes in sensor inputs, processes data, and sends instructions through a Digital Analog Port to the Motor Controller. It is also directly powered from the battery [1].
- **Motor System**: Comprises Left and Right Motors controlled by a Motor Controller. The controller divides power and executes movement commands based on signals from the Controller Unit [1].

This modular structure ensures stable communication and control across all components, forming a complete autonomous vehicle system [1].

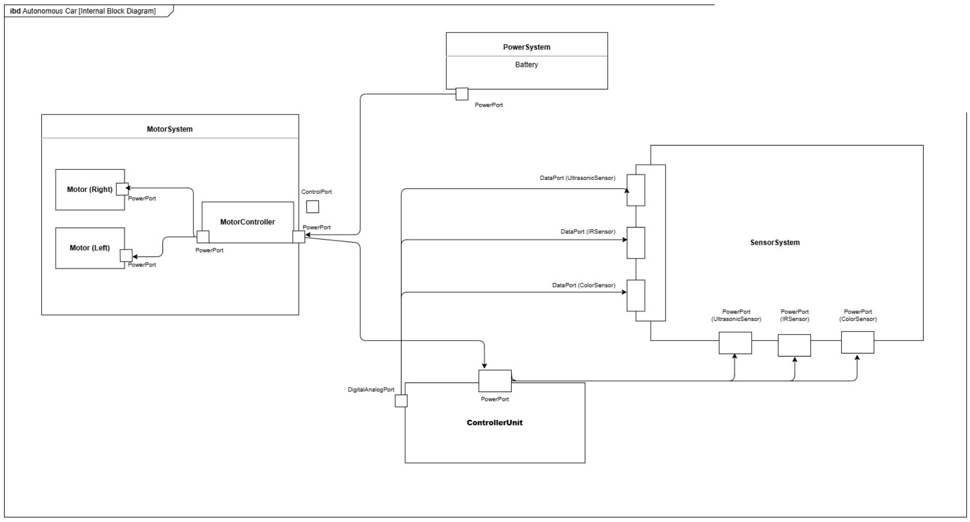

Figure 4: SysML Internal Block Diagram of the Autonomous Vehicle

2.5 Block Definition Diagram (BDD) Overview

Figure 5 shows a Block Definition Diagram (BDD) that provides post-level structure and decomposition of the autonomous vehicle system using five former subsystems. These are the contained subsystems of the autonomous vehicle system as shown in the BDD: Motor System, Power System, Controller Unit, Sensor System, and Car Body. Each subsystem contains functions, activities, or components needed to accomplish vehicle autonomy [1].

- **Motor System**: This subsystem contains two motors (left and right) and a motor controller. The motor system will define the movement of the vehicle for backwards, forwards, and directional. The system also defines controlled operations for control of the individual motors and defines important parameters such as maximum RPM [1].

- **Power System**: This subsystem provides electrical power to all of the subsystems of the vehicle via a LiPo battery. The system defines the system-wide voltage and includes ports for charging and power distribution [1].

- **Controller Unit**: This subsystem represents the main logic unit (Arduino Uno). This is the central logic unit that takes inputs from the sensors and executes control algorithms. It defines properties including clock speed, memory, and voltage, and defines ports for I/O and communication. The controller also defines operations, such as hardware initialization and where the main-loop command is executed [1].

- **Sensor System**: The sensor system contains multiple sensors including two IR sensors to track the line, a color sensor to see the color of obstacles, and an ultrasonic sensor to measure distances. It provides sensor data to the controller through a single data output port, and implements functionality to read sensor values [1].

- **Car Body**: The car body represents the mechanical nature of the vehicle, including the frame, wheels, servo motor, and mounting positions. It captures attributes associated with the car body including weight, size, and material, and it provides standard ports to connect the motors, sensors, and power system [1].

The modular design offers a clear separation of functions, and reduces the complexity of integrating the system while improving ease of maintaining and scaling the autonomous vehicle design [1].

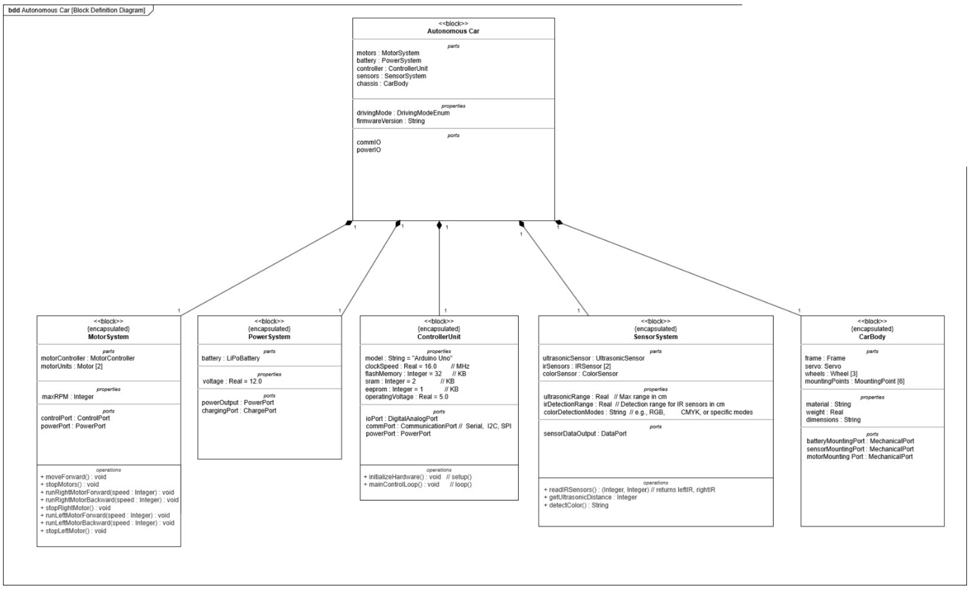

Figure 5: SysML Block Definition Diagram (BDD) of the Autonomous Vehicle

2.6 Use Case Diagram

The Use Case Diagram shows the way the user interacts with the system of the autonomous car. The user instantiates the Start Vehicle use case, and the core actions like Control Movement and Follow Line can occur with coordination of the motor and IR sensor. The Detect Obstacle use case, enabled by the ultrasonic sensor, will be invoked by both routine, line following or stopping. Once the obstacle is detected the system goes into the Stop Vehicle use case to avoid collision. This diagram covers the key functions of the system and their dependencies on sensors and actuators [1].

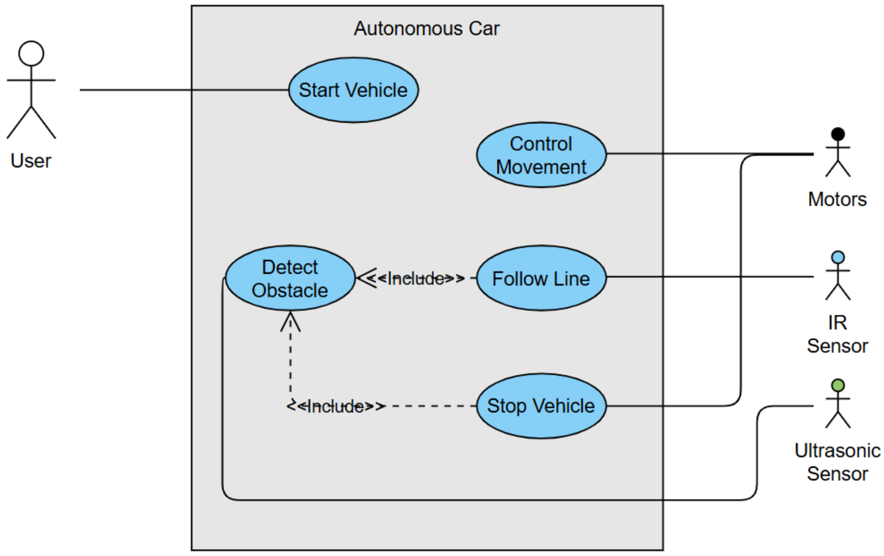

Figure 6: SysML Use Case Diagram of the Autonomous Vehicle

2.7 Sequence Diagram

The sequence diagram illustrates the interaction between a user, the vehicle control system, and the colour sensor. The process begins with the colour sensor sending colour data to the vehicle control system. Based on the received data, the system performs three checks and prints the result to the serial monitor. If the detected colour is blue or red, it prints "Blue" or "Red" respectively and confirms detection. If no specific colour is detected, it prints "any colour" and concludes with "NO colour detected." This looped analysis helps ensure reliable colour identification and response within the autonomous system [1].

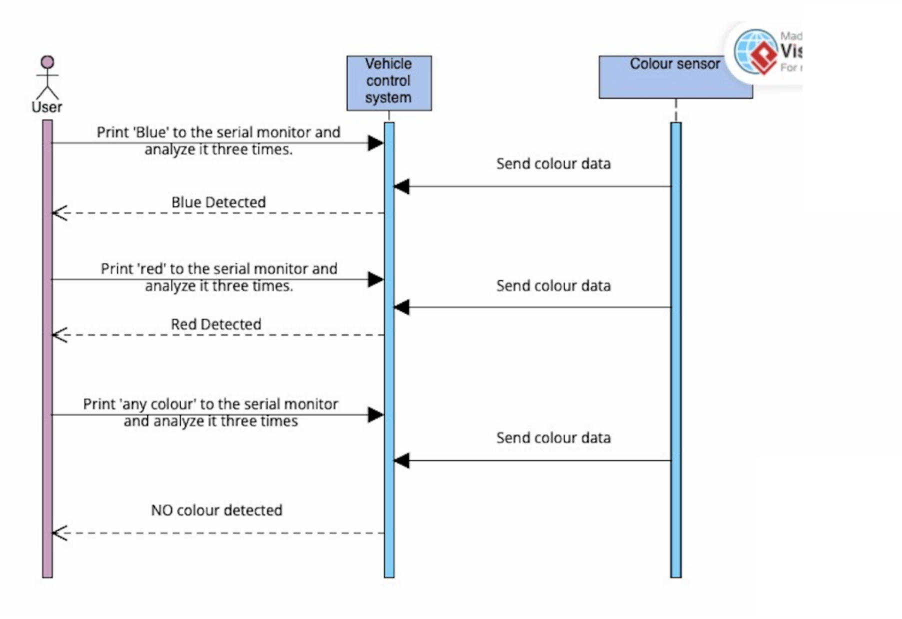

Figure 7: SysML Sequence Diagram for Color Detection

The sequence diagram illustrates the interaction between a user, a line-following car, its control system, and IR sensors. The system uses left and right IR sensors to detect the vehicle's alignment on the track. If the right sensor detects deviation, the system prints "moving left" on the serial monitor and adjusts the car to the left to maintain alignment. Similarly, if the left sensor is triggered, the system prints "moving right" and shifts the vehicle slightly to the right. This constant data exchange between sensors and the vehicle control system enables smooth and continuous path correction for effective line following [1].

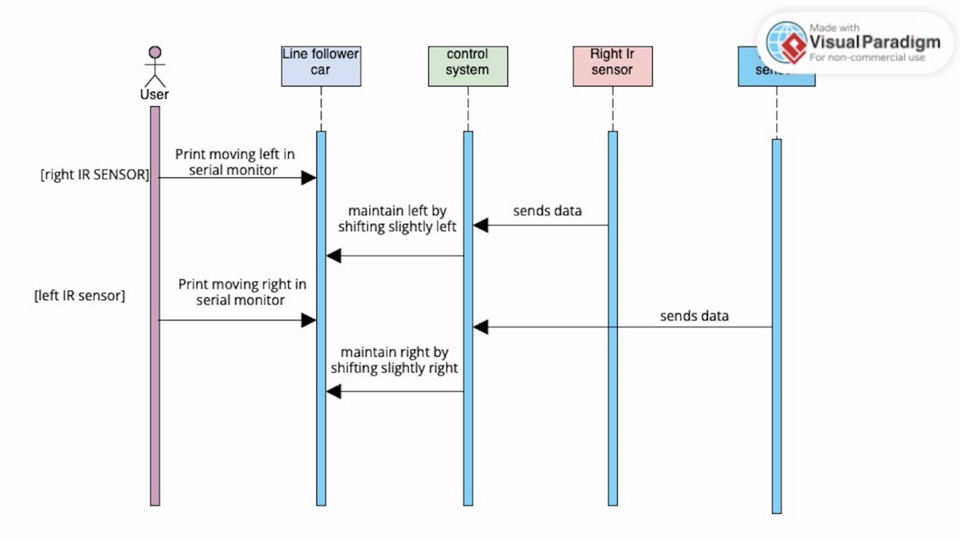

Figure 8: SysML Sequence Diagram for Line Following

3 Structural Component Design

Figure 9 illustrates three key structural components designed and fabricated to ensure the secure mounting of critical modules within the autonomous vehicle prototype. All designs were created using CAD tools such as SolidWorks and Tinkercad, then 3D printed using PLA or laser-cut from wood depending on the required strength and positioning [1].

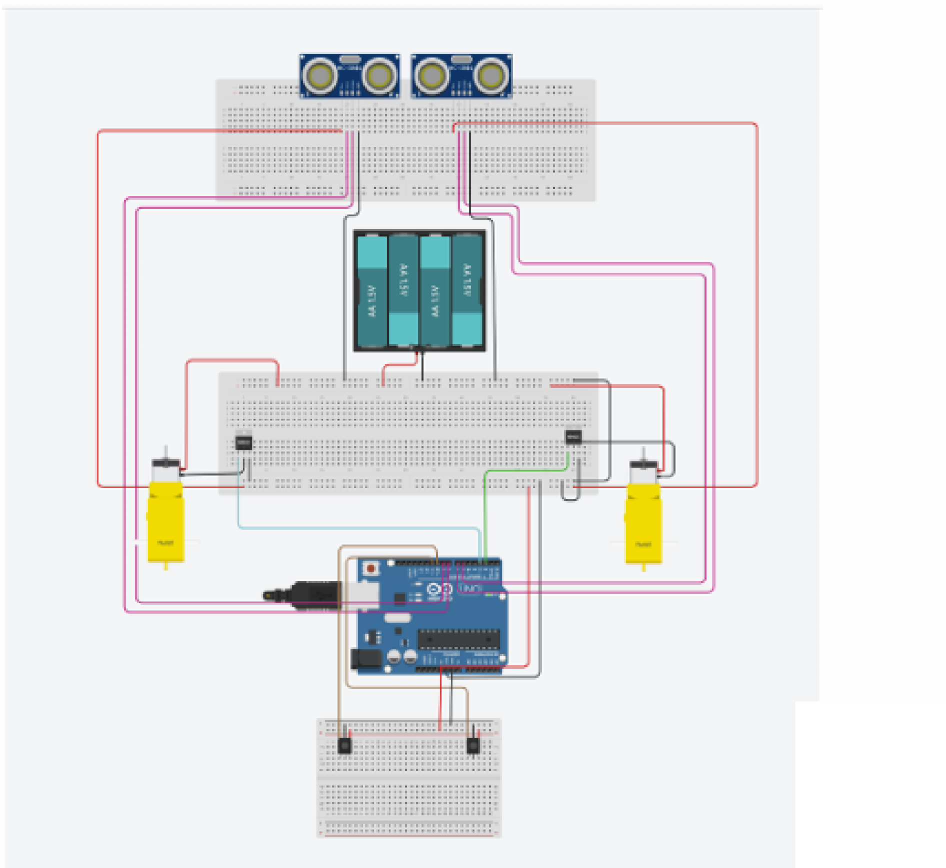

Figure 9: Key Structural Components of the Autonomous Vehicle

3.1 Battery Holder

The largest structure in the design layout serves as the battery holder, crafted to accommodate a standard LiPo battery. The dimensions were tailored to snugly fit the 7.4V 3200mAh unit while allowing cable clearance. Mounting holes and slots were integrated to ensure easy fastening to the vehicle base, while still allowing for efficient replacement and recharge operations. Its position on the main chassis was carefully selected to maintain center-of-gravity balance [1].

3.2 IR Sensor Holder

The component shown on the left (top corner) in Figure 9 is the IR sensor holder, designed to house the ST1140 IR module. It includes a precise cutout and screw holes to secure the sensor tightly in place. The holder tilts the IR sensor at an optimized downward angle to ensure maximum surface detection accuracy. This ensures stable performance during line following, especially when tracking black lines on white backgrounds [1].

3.3 Motor Holder

The component shown in the bottom-left corner of Figure 9 represents the motor holder. It was designed to tightly fit standard 5V-9V DC motors and features side flanges with 2.5 mm screw holes for stable attachment to the wooden base. The motor shaft alignment was carefully considered in the design to avoid slippage and ensure even torque delivery to the rear wheels, enhancing driving stability. All parts were designed for modularity and ease of assembly, contributing to a more maintainable and adjustable prototype [1].

4 Components

A variety of hardware components were integrated into the autonomous vehicle prototype. Each component played a specific role in ensuring the successful execution of functions such as motion control, obstacle detection, line following, and colour recognition.

4.1 Arduino UNO

The Arduino Uno serves as the brain of the system, interfacing with all sensors and actuators. It features the ATmega328P microcontroller, an 8-bit RISC core, 14 digital GPIO pins, and 6 analog input pins. It processes incoming data and controls motor drivers accordingly [1].

4.2 LiPo Battery

A 7.4V 3200 mAh 20C LiPo battery from Conrad was used as the power source for the L298N motor driver and motors. With a compact design and 210 g weight, it provides sufficient discharge current for all driving components [1].

4.3 9V DC Motors

Two 9V brushed DC motors were installed to drive the rear wheels of the prototype. These motors offer sufficient torque and speed for small-scale autonomous navigation. They were powered through the L298N driver instead of the Arduino to meet current demands [1].

4.4 L298N Motor Driver

The L298N module was used to drive the DC motors and supply power to the Arduino. It accepts power from the LiPo battery and provides direction and speed control via PWM signals from the Arduino. The module also ensures shared ground and voltage regulation [1].

4.5 HC-SR04 Ultrasonic Sensor

The HC-SR04 module was used for real-time distance measurement between 2 cm and 3 m, ideal for obstacle detection. It uses 40 kHz ultrasonic pulses and reflects them back to compute the distance. The sensor provides input to the system to decide when to stop or reroute [1].

4.6 ST1140 IR Sensors

Two ST1140 IR sensors were mounted at the base to follow a black line on a light surface. Each sensor emits infrared light and detects reflections to determine if the surface is reflective (white) or absorbing (black). This forms the basis of the vehicle's path-following logic [1].

5 Code Explanation

This section provides a detailed explanation of the Arduino code implemented for the autonomous line-following vehicle. The code orchestrates the interaction between various sensors (IR, ultrasonic, servo) and actuators (DC motors, servo motor) to enable line following, obstacle avoidance, and re-orientation capabilities.

```
#include <Servo.h>

// === Pin Definitions ===
#define ENA 9   // Right motor speed
#define IN1 8   // Right motor direction
#define IN2 7   // Right motor direction

#define ENB 4   // Left motor speed
#define IN3 6   // Left motor direction
#define IN4 5   // Left motor direction

#define IR_LEFT 3
#define IR_RIGHT 2

#define TRIG_PIN 12
#define ECHO_PIN 13

const int SERVO_PIN = 10;
const int SCAN_CENTER = 0;
const int SCAN_RIGHT = 90;
const int SCAN_LEFT = 180;

// === Adjustable speed ===
int motorSpeed = 65;           // base forward speed
float backwardsScale = 1;      // scale for backward motor speed (0-1)
int angleSnapSpeed = 75;       // turning speed for 90 degree turns
int obstacleThreshold = 15;    // obstacle distance threshold in cm

Servo scanServo;

void setup() {
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(ENA, OUTPUT);
```

```

  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  pinMode(ENB, OUTPUT);

  pinMode(IR_LEFT, INPUT);
  pinMode(IR_RIGHT, INPUT);

  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);

  scanServo.attach(SERVO_PIN);
  scanServo.write(SCAN_CENTER);

  stopMotors();
}

void loop() {
  if (measureDistance() < obstacleThreshold) {
    avoidObstacle();
    return;
  }

  int leftIR = digitalRead(IR_LEFT);
  int rightIR = digitalRead(IR_RIGHT);

  if (leftIR == LOW && rightIR == LOW) {
    moveForward();
  }
  else if (leftIR == HIGH && rightIR == HIGH) {
    stopMotors();
  }
  else {
    if (leftIR == LOW && rightIR == HIGH) {
      runRightMotorForward(motorSpeed);
      runLeftMotorBackward(int(motorSpeed * backwardsScale));
    }
    else if (leftIR == HIGH && rightIR == LOW) {
      runLeftMotorForward(motorSpeed);
      runRightMotorBackward(int(motorSpeed * backwardsScale));
    }
  }
}

// === Obstacle Avoidance ===
void avoidObstacle() {
  stopMotors();
  delay(500);

  // Step 1: turn left to get off the line
  runLeftMotorBackward(angleSnapSpeed);
  runRightMotorForward(angleSnapSpeed);
  delay(400);

  // Step 2: move forward out of the way
  moveForward();
  delay(600);

```

```
  // Step 3: turn right
  runLeftMotorForward(angleSnapSpeed);
  runRightMotorBackward(angleSnapSpeed);
  delay(400);

  // Step 4: move forward across the obstacle
  moveForward();
  delay(800);

  // Step 5: turn right again
  runLeftMotorForward(angleSnapSpeed);
  runRightMotorBackward(angleSnapSpeed);
  delay(400);

  // Step 6: move forward back toward the line
  moveForward();
  delay(600);

  // Step 7: turn left to face original direction
  runLeftMotorBackward(angleSnapSpeed);
  runRightMotorForward(angleSnapSpeed);
  delay(400);

  stopMotors();
  delay(500);

  reorientAfterAvoidance();
}

// === Reorient using Servo ===
void reorientAfterAvoidance() {
  stopMotors();
  delay(500);

  int frontDistance, rightDistance, leftDistance;

  scanServo.write(SCAN_RIGHT);
  delay(500);
  rightDistance = measureDistance();

  scanServo.write(SCAN_CENTER);
  delay(500);
  frontDistance = measureDistance();

  scanServo.write(SCAN_LEFT);
  delay(500);
  leftDistance = measureDistance();

  scanServo.write(SCAN_CENTER);
  delay(300);

  if (frontDistance < rightDistance && frontDistance < leftDistance) {
    turnRight90();
    delay(300);
    turnRight90();
  } else if (rightDistance < frontDistance && rightDistance <
   leftDistance) {
    turnLeft90();
```

```
  } else if (leftDistance < frontDistance && leftDistance < rightDistance) {
    turnRight90();
  }
  stopMotors();
  delay(500);
}

// === Turning Helpers ===
void turnLeft90() {
  runLeftMotorBackward(angleSnapSpeed);
  runRightMotorForward(angleSnapSpeed);
  delay(400);
  stopMotors();
  delay(300);
}

void turnRight90() {
  runLeftMotorForward(angleSnapSpeed);
  runRightMotorBackward(angleSnapSpeed);
  delay(400);
  stopMotors();
  delay(300);
}

// === Motor Control ===
void moveForward() {
  runRightMotorForward(motorSpeed);
  runLeftMotorForward(motorSpeed);
}

void stopMotors() {
  stopLeftMotor();
  stopRightMotor();
}

void runRightMotorForward(int speed) {
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, speed);
}

void runRightMotorBackward(int speed) {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  analogWrite(ENA, speed);
}

void stopRightMotor() {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, 0);
}

void runLeftMotorForward(int speed) {
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  analogWrite(ENB, speed);
```

```
}

void runLeftMotorBackward(int speed) {
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  analogWrite(ENB, speed);
}

void stopLeftMotor() {
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, LOW);
  analogWrite(ENB, 0);
}

// === Ultrasonic ===
long measureDistance() {
  digitalWrite(TRIG_PIN, LOW);
  delayMicroseconds(2);
  digitalWrite(TRIG_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIG_PIN, LOW);
  long duration = pulseIn(ECHO_PIN, HIGH);
  long distance = duration * 0.034 / 2;
  return distance;
}
```

Listing 1: Arduino Code for Autonomous Line-Following Car

5.1 Pin Definitions and Global Parameters

The initial section of the code defines constants and global variables that link the software to the physical pins of the Arduino Uno and the components they control. This provides a clear interface for hardware-software interaction.

- `#define` constants: These map specific Arduino digital pins to motor control (ENA, IN1, IN2, ENB, IN3, IN4), IR sensors (IR_LEFT, IR_RIGHT), and the ultrasonic sensor (TRIG_PIN, ECHO_PIN).
- `const int SERVO_PIN`: Defines the digital pin for the servo motor.
- `const int SCAN_CENTER, SCAN_RIGHT, SCAN_LEFT`: These define the servo angles for scanning different directions (0, 90, and 180 degrees respectively).

The adjustable global variables are crucial for tuning the robot's behavior:

Table 1: Key System Parameters and Their Roles

Parameter	**Default Value**	**Unit**	**Description / Role**
`motorSpeed`	65	(0-255 PWM)	Base speed for forward movement.
`backwardsScale`	1.0	(0-1.0)	Scaling factor for backward motor speed.
`angleSnapSpeed`	75	(0-255 PWM)	Motor speed used for 90-degree turns.
`obstacleThreshold`	15	cm	Distance threshold for obstacle detection.
`SCAN_CENTER`	0	degrees	Servo angle for scanning straight ahead.
`SCAN_RIGHT`	90	degrees	Servo angle for scanning to the right.
`SCAN_LEFT`	180	degrees	Servo angle for scanning to the left.

These parameters highlight the empirical nature of robotic development, where optimal performance often depends on careful calibration to specific hardware and environmental conditions [1].

5.2 The `setup()` Function

The `setup()` function is executed once at the start of the program, initializing all hardware components. It configures the digital pins connected to the motor driver (IN1, IN2, ENA, IN3, IN4, ENB) as outputs, and the IR sensor pins (IR_LEFT, IR_RIGHT) and ultrasonic sensor pins (TRIG_PIN, ECHO_PIN) as inputs. The servo motor is attached to its designated pin (`SERVO_PIN`) and initialized to the center position (`SCAN_CENTER`). Finally, `stopMotors()` is called to ensure the robot is stationary at startup. This function directly corresponds to the "Initialization" state in the SysML State Machine Diagram (Figure 2) [1].

5.3 The `loop()` Function: The Main Control Loop

The `loop()` function contains the continuous execution cycle of the robot's primary behaviors. It dictates the autonomous operation, starting with an obstacle detection check.

- **Obstacle Detection**: The `measureDistance()` function is called. If the measured distance is less than `obstacleThreshold` (15 cm), the `avoidObstacle()` function is invoked, and the current loop cycle is exited. This implements the "ObstacleDetection" state (Figure 2) [1].
- **Line Following Logic**: If no obstacle is detected, the robot proceeds with line following. The states of the left and right IR sensors are read.
  - If both sensors are `LOW` (on the black line), `moveForward()` is called.
  - If both sensors are `HIGH` (off the line), `stopMotors()` is called, possibly indicating the end of the line.
  - If only one sensor is off the line, the robot adjusts its direction:
    - If `rightIR` is `HIGH` (right sensor off line), the robot turns right by running the right motor forward and the left motor backward.
    - If `leftIR` is `HIGH` (left sensor off line), the robot turns left by running the left motor forward and the right motor backward.

This logic directly implements the "LineFollowing" state and the decision logic for curve/obstruction handling in the SysML Activity Diagram (Figure 3) [1].

5.4 The `avoidObstacle()` Function

The `avoidObstacle()` function executes a pre-programmed sequence of movements to navigate around a detected obstacle. It involves a series of fixed-duration movements:

1. Stop motors, then delay.
2. Turn left to move off the line (left motor backward, right motor forward for 400ms).

3. Move forward to clear the path (600ms).
4. Turn right (400ms).
5. Move forward across the obstacle's original position (800ms).
6. Turn right again (400ms).
7. Move forward back toward the line (600ms).
8. Turn left to re-align with the original direction (400ms).

Each step relies on fixed `delay()` values, which can lead to inaccuracies due to varying factors like battery level or surface friction. This routine implements the "Go around the obstacle" action in the SysML Activity Diagram (Figure 3) [1]. After this sequence, `reorientAfterAvoidance()` is called to correct for accumulated errors.

5.5 The `reorientAfterAvoidance()` Function

The `reorientAfterAvoidance()` function re-aligns the robot using real-time sensor feedback after the fixed obstacle avoidance routine.

- **Environmental Scan**: The `scanServo` is commanded to `SCAN_RIGHT`, `SCAN_CENTER`, and `SCAN_LEFT` positions, taking distance readings (`rightDistance`, `frontDistance`, `leftDistance`) at each point using `measureDistance()`.
- **Decision Logic**: The robot determines the optimal re-orientation:
  - If `frontDistance` is the smallest, the robot performs two consecutive 90-degree right turns (180-degree turn).
  - If `rightDistance` is the smallest, it performs a 90-degree left turn.
  - If `leftDistance` is the smallest, it performs a 90-degree right turn.

This function implements the "CalculateAlternatePath" state in the SysML State Machine Diagram (Figure 2) [1]. It provides an adaptive layer to compensate for the inaccuracies of time-based movements.

5.6 Turning Helper Functions (`turnLeft90()`, `turnRight90()`)

These functions achieve approximate 90-degree turns using a differential drive mechanism (one motor forward, one backward) at a speed defined by `angleSnapSpeed`. A fixed `delay(400)` milliseconds controls the turn duration. These functions directly implement the "Turn 90 Degrees" requirement (ID=009) [1].

5.7 Motor Control Functions

The code includes low-level functions for precise motor control, interfacing with the L298N motor driver:

- `moveForward()`: Sets both motors to run forward at `motorSpeed`.
- `stopMotors()`: Halts both motors.

- `runRightMotorForward(int speed)`, `runRightMotorBackward(int speed)`, `stopRightMotor()`: Control the direction and speed of the right motor using digital and analog writes to IN1, IN2, and ENA pins.
- `runLeftMotorForward(int speed)`, `runLeftMotorBackward(int speed)`, `stopLeftMotor()`: Control the direction and speed of the left motor using digital and analog writes to IN3, IN4, and ENB pins.

These functions provide the fundamental actuation capabilities, mapping to the "Motor System" in the SysML Internal Block Diagram (Figure 4) [1].

5.8 Ultrasonic Measurement (`measureDistance()`)

This function obtains distance readings from the HC-SR04 ultrasonic sensor. It sends a 10-microsecond `HIGH` pulse to `TRIG_PIN`, measures the duration of the `HIGH` pulse on `ECHO_PIN` (time for sound to travel and return), and calculates distance using the formula `duration * 0.034 / 2` (0.034 cm/microsecond is the speed of sound, divided by 2 for round trip). This function is a critical part of the "Sensor System" in the SysML diagrams [1].

6 Experimental Results

The project successfully culminated in the development of a line-following autonomous vehicle capable of making independent decisions based on the color of detected obstacles. The experimental results, as stated in the project's conclusion, demonstrate that the prototype successfully fulfills all primary functional requirements [1]. While specific quantitative data, such as success rates on complex tracks, precision of turns, or detailed obstacle avoidance success rates, are not explicitly provided within the available documentation, the assertion of successful fulfillment indicates that the system performed as intended during testing.

The project's primary objective appears to have been the validation of the system's *capability* to perform the specified tasks—line following, obstacle avoidance, and reorientation—rather than achieving optimized or benchmarked performance metrics. This functional validation is a crucial first step in academic and prototyping projects, establishing a proof-of-concept for the integrated system. For future industrial applications or more rigorous scientific studies, the inclusion of detailed quantitative performance data would be essential for comparative analysis and precise identification of areas for improvement.

7 Conclusion

This report has detailed the comprehensive development of an autonomous line-following vehicle, designed with the advanced capability of making independent decisions based on the color of detected obstacles. The project successfully integrated mechanical design, electronic components, and sophisticated software logic to create a functional prototype. The systematic approach, guided by SysML, ensured a structured development process, from defining granular requirements to modeling complex system behaviors. The experimental outcomes affirm that the prototype successfully meets all its primary functional requirements, demonstrating its ability to navigate lines, detect obstacles, and react intelligently [1].

Looking ahead, several key enhancements are envisioned to further advance the vehicle's capabilities and robustness. To significantly improve stability and maneuverability, future iterations plan to integrate two additional DC motors, replacing the current passive ball bearing front wheel with a more controlled drive system [1]. This upgrade would allow for more precise steering and better traction, especially on varied surfaces. Furthermore, the sensing capabilities will be enhanced by mounting an additional HC-SR04 ultrasonic sensor at the rear of the vehicle. Both the front and rear ultrasonic sensors will be attached to a servo motor, enabling dynamic 360-degree environmental scanning. This expanded perception will facilitate more consistent and informed decision-making, moving beyond simple front-facing obstacle detection [1]. To support these substantial hardware upgrades, a critical step will involve adopting a microcontroller with an increased number of digital and analog I/O pins, as the current Arduino Uno would likely reach its I/O and processing limits with the added complexity [1]. These proposed enhancements demonstrate a clear understanding of the prototype's current limitations and a strategic vision for evolving towards a more capable, situationally aware, and truly autonomous robotic system.

8 Acknowledgment

The successful completion of this project was made possible through the invaluable support and resources generously provided by Hamm-Lippstadt University of Applied Sciences [1]. The project team extends its sincere gratitude to Prof. Dr. S. Henkler and Prof. G. Wahrmann for their exceptional guidance, continuous encouragement, and constructive feedback throughout every phase of the project [1].

9 Appendix

All team members contributed equally to the design, development, and successful completion of this project [1].

References

References

[1] Joy-IT. "Color sensor module TCS3200." Joy-IT.net. Available: https://joy-it.net/de/products/SEN-Color. [Accessed: 21-Jun-2025].

[2] Santos, R. "Complete Guide for Ultrasonic Sensor HC-SR04 with Arduino." Random Nerd Tutorials. Available: https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/. [Accessed: 21-Jun-2025].

[3] Lextronic. "Capteur de ligne Arduino OpenST1140." Lextronic.fr. Available: https://www.lextronic.fr/capteur-ligne-arduino-openst1140-51718.html. [Accessed: 21-Jun-2025].

[4] Components101. "L293D Motor Driver Module." Components101.com. Available: https://components101.com/modules/l293n-motor-driver-module. [Accessed: 21-Jun-2025].

[5] Arduino. "Arduino UNO Rev3 with long pins (Retired)." Arduino Documentation. Available: https://docs.arduino.cc/retired/boards/arduino-uno-rev3-with-long-pins/. [Accessed: 21-Jun-2025].

[6] Autodesk. "Tinkercad Circuits: Online Simulator for Arduino and Electronics." Tinkercad.com. Available: https://www.tinkercad.com. [Accessed: 21-Jun-2025].