Peacekeeper Bot

by
Adam Kearsey
Evan Sawchuk
Jasvinder Dhaliwal

Apr 3, 2025

Table of Contents

1 Introduction..... 1
1.1 Literature Review..... 1
1.2 Goals..... 2
2 Methodology..... 3
2.1 Overview..... 3
2.2 Application..... 3
2.2.1 Hardware..... 3
2.2.2 Software..... 5
3 Results..... 8
3.1 Testing..... 8
3.2 Conclusion..... 12
3.3 Future Work..... 12
References..... 14
Appendices..... 16
Appendix - A: Code..... 16

1 Introduction

1.1 Literature Review

Robots are playing an increasingly important role in military and law enforcement work, doing tasks like gathering environmental data or even supporting combat missions. The success of these autonomous machines depends heavily on their ability to move through their surroundings and adapt to sudden changes. As a result, researchers have focused on improving how robots see, move, and make decisions using techniques like SLAM (Simultaneous Localization and Mapping), sensor fusion, and reinforcement learning.

Zhao et al. [15] tackled the problem of unreliable GPS in farming environments by developing the EALVIO system, which combines SLAM with flexible sensor use depending on how complex the environment is. This method made location tracking more accurate while using less computing power. Their work connects with Ramezani et al. [9], who proposed using reinforcement learning along with UAVs to help with navigation and decision-making. Together, these approaches show how SLAM and learning-based methods can work side by side to improve real-time performance.

Liang et al. [6] added to this by focusing on tracking moving targets with AprilTags, which allow for precise position tracking as long as the tags are visible. When the tags couldn't be seen, a backup color-tracking system was used to keep the target in view. This idea complements Ramezani et al.'s [9] UAV-assisted navigation and echoes the challenges highlighted by Hu and Assaad [5], who found that vision-based systems often fail in poor weather or lighting. These studies point to the need for more resilient tracking methods that work across different environmental conditions.

Liu et al. [7] explored how using several types of sensors together—both external and internal—can help robots better understand their surroundings. Their multi-sensor approach supports the work of Zhao et al. [15] and Ramezani et al. [9] by offering a more flexible way to handle changes in the environment. Liu et al. also looked into deep learning to improve what robots "see" and "understand," which ties in with the tracking problems discussed by Liang et al. [6].

Wong et al. [14] designed a pathfinding algorithm called Fuzzy-Obstacle-Avoidance-Path that helps robots avoid obstacles in real time. This kind of adaptive navigation could be a good match for SLAM and reinforcement learning systems, though their testing was limited. That lack of real-world validation is something many vision-based systems struggle with.

Megalingam et al. [8] created an autonomous wall-painting robot that uses ultrasonic sensors for navigation. Even though it was meant for indoor use, their method offers useful ideas for outdoor robots as well—especially when combined with sensor fusion, as Liu et al. [7] suggested. Similarly, Cheng et al. [1] worked on a robot that could shoot basketballs by using a camera to measure distance. While it worked under controlled

conditions, relying on a single camera showed limitations that multi-sensor setups could fix.

Gogoi et al. [4] and Tatar et al. [13] built turret systems for firefighting and stability control, respectively. Their work ties in with that of Tanyildizi et al. [12] and Colon et al. [2], who focused on fast, accurate target detection and tracking. These studies all point to a shared goal: improving speed, accuracy, and control in environments where targets can move unpredictably. Many of the tools used—like SLAM, AprilTags, and sensor fusion—can be combined to build smarter and more responsive systems.

Altogether, this research shows how the field is moving from basic navigation toward fully integrated systems that can find, follow, and respond to targets on their own. The combination of SLAM, reinforcement learning, multi-sensor input, and precise control systems offers a strong path forward for building intelligent, autonomous robots.

Still, despite this progress, key challenges remain. Zhao et al. [15] noted that processing sensor data can be computationally expensive, making it hard to scale these systems to larger areas. Ramezani et al. [9] also pointed out that reinforcement learning is very resource-intensive and may not be fast enough for real-time use. Liang et al. [6] relied on AprilTags, which only work if visible, and their backup color-tracking system can be inaccurate. Hu and Assaad [5] showed that vision systems can fail under difficult lighting or weather. Liu et al. [7] proposed multi-sensor systems, but these remain mostly theoretical when it comes to extreme environments. Wong et al. [14] didn't test their system extensively in real-world settings, and Megalingam et al. [8] required manual setup, reducing the level of autonomy. Cheng et al. [1] used targets of a fixed size, which doesn't reflect the variety of shapes and sizes seen in practice. Gogoi et al. [4] and Tatar et al. [13] only tested their systems on unmoving targets, limiting their real-world usefulness. Tanyildizi et al. [12] found that their turret could mistakenly engage any moving object, increasing the risk of false positives. Colon et al. [2] achieved a 100 ms targeting response, which might not be fast enough for very quick targets.

This project aims to add to the existing research by trying to deploy a multisensor color and object-detecting robot using only basic robotics technologies and simplified detection algorithms.

1.2 Goals

The goal of the project is to create an Unmanned Ground Vehicle that identifies both ground and air entities and responds accordingly.

2 Methodology

2.1 Overview

After reviewing the existing research, the group weighed all the options to deliver a fully automated turret robot. Considerations were made to the time constraints, resources available, and previous expertise of the group. Since the group had prior experience with image recognition and classification, the idea of shooting a moving target was intriguing.

The firing mechanism was deliberated, considering Nerf bullets feeding into a flywheel system that spun the Nerf bullet into a barrel, or a fully functioning Nerf pistol that fired upon a trigger pull. Due to the potential issue of the flywheels not spinning fast enough to hit distant targets, the choice was the Nerf pistol.

The robot does not require any mapping functionality, considering it will shoot a target once it comes across it, so a simple object-avoidance function would suffice. A graphical representation of the preliminary design is seen in Fig. 6.

2.2 Application

2.2.1 Hardware

The sensors to be used in this project are the VL530X distance sensor [11] and the DFRobot FIT0701 USB Camera [3]. These sensors are necessary to navigate the environment and identify specific targets while avoiding obstacles in the vehicle's path. The distance sensor will detect obstructions in the direct path of the robot's environment, and the camera will assist in the identification of targets and aiming accuracy.

As shown in Table I, the VL53L0X is capable of detecting distances of 100cm, which is the threshold for path obstruction detection. This distance gives the robot time and space clearance to turn around or make a decision on navigating obstacles.

TABLE 1: Specifications of components used

components	Specs
VL53L0X distance sensor	Range: ~30 to 1000mm Size: 21.0mm x 18.0mm x 2.8mm Interface: I2C Min/max operating voltage: 2.6v/5.5v Supply current 20mA

DFRobot FIT0701 USB Camera [3]	0.3 MegaPixels USB 2.0 70-degree FOV 640x480 resolution Compact size 30x25x21.4mm 5V DC via USB Operating Temperature -20 to 70°C Current unknown as spec sheets only mention 5V power input
Sabertooth 12A Motor Driver [16]	Input Voltage: 6V to 25V (2s to 6s lithium) Output current: Up to 12A per motor Peak Output current: Up to 25A per motor Weight: 1.5oz/43 grams Size: 2.35” x 1.85” x .6”
Hitec HSS-422 servo motor	Operating voltage range: 4.8V TO 6.0V Min Stall torque:3.3kg.cm @ 4.8v Size: 40.6x19.8x36.6mm

The DFRobot FIT0701 USB Camera has a 640×480 resolution and ranges up to a 1280x960 resolution, 70° FOV, and 0.3 megapixels for image identification. It is also compact and meets the Raspberry Pi’s voltage requirement of 5 V. Both sensors will be connected via USB to either a dual USB port Raspberry Pi [10] or a dual port as seen in Fig. 1. The Raspberry Pi 4 provides 1.6 A of power to downstream USB peripherals when connected to a power supply capable of 3 A at +5 V (15 W). When connected to any other compatible power supply, the Raspberry Pi 4 restricts downstream USB devices to 600 mA of power.

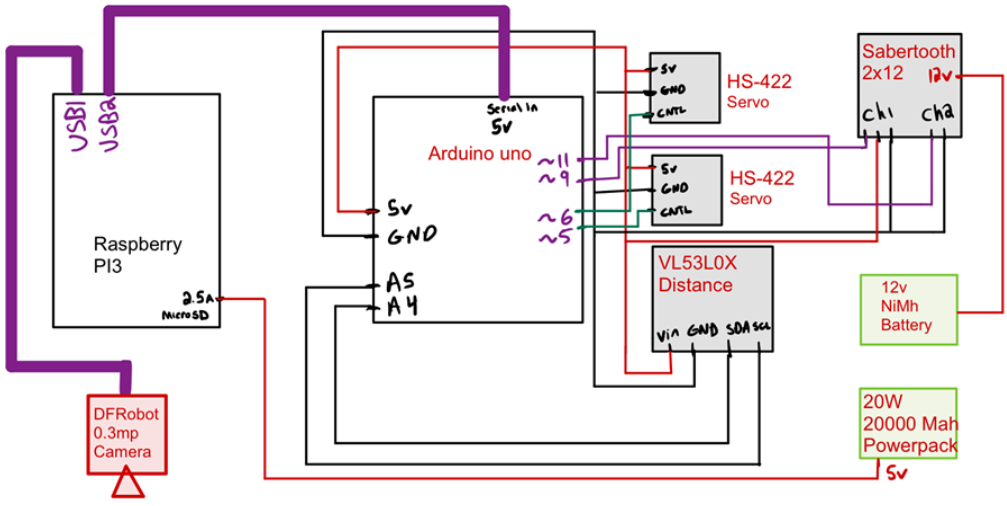


Fig. 1. Wiring diagram

The Robot used two Hitec HSS-422 servo motors for the operation of the Nerf gun. One servo motor was mounted to the gun, and the arm was attached to the slide pin of the gun. When this motor was powered, it would rotate the arm and in return, push the bullet into the flywheels and afterwards return to its original position to allow the next bullet to be loaded. The second servo motor was used to operate the switch that sent power to the flywheels. The arm on the motor would rotate down onto a physical switch for a couple of seconds to allow the flywheels to reach maximum speed, and once the bullet had been shot, the servo's arm would retract, and the flywheels would slow down.

The VL53L0X distance sensor operates using a time-of-flight (ToF) principle to measure absolute distance by calculating how long it takes for emitted infrared light to reflect off a surface and return to the sensor [11]. It communicates over the I²C protocol, which allows for simple two-wire communication (SDA and SCL).

Structurally, a large robot chassis with four rubber wheels was used to build the robot. In addition to this chassis, a 3D printed structure was also necessary to house the gun and camera. Originally, the structure was designed with a LIDAR sensor in mind, so the structure has four holes to allow the prongs of a LIDAR to sit flush with the surface of the structure. Later, the LIDAR was switched out for a distance sensor, and the gun was mounted on top of the structure to account for any clearance issues. You can see the 3D renders in Fig. 2 and Fig. 3, where Fig. 2 is the flat surface of the structure and is held up by four units of the columns in Fig.3.

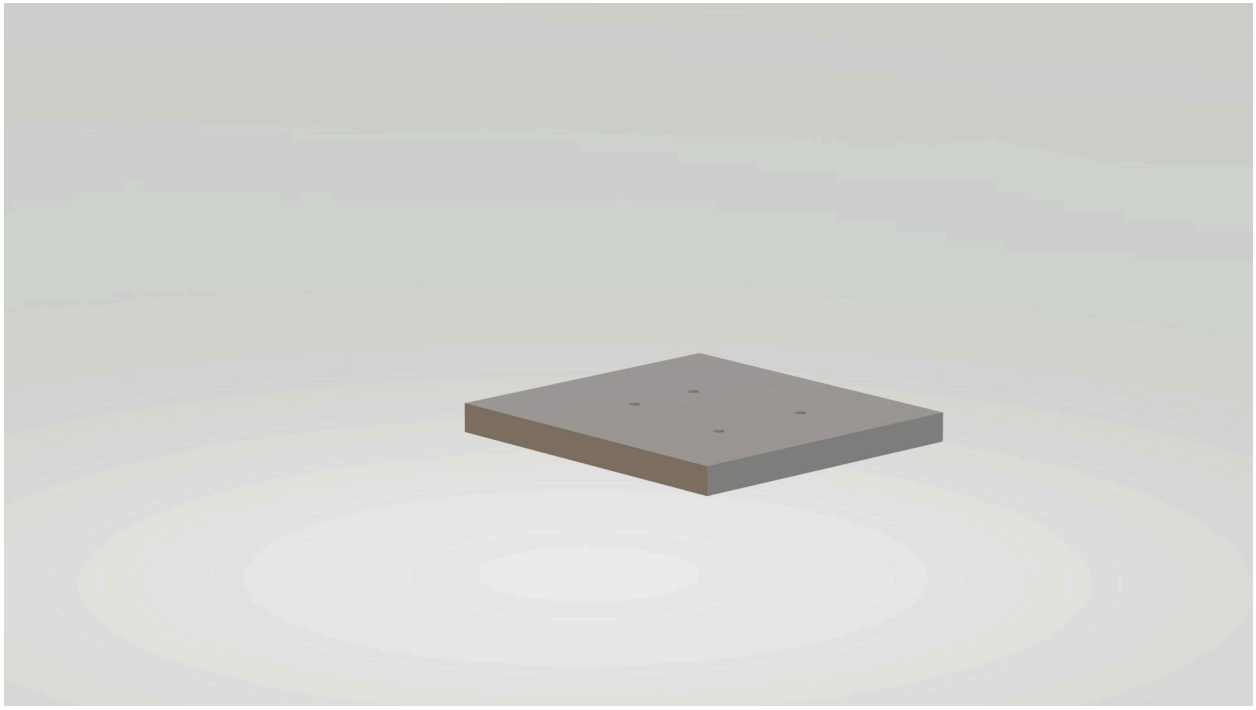


Fig. 2. 3D Printed Base

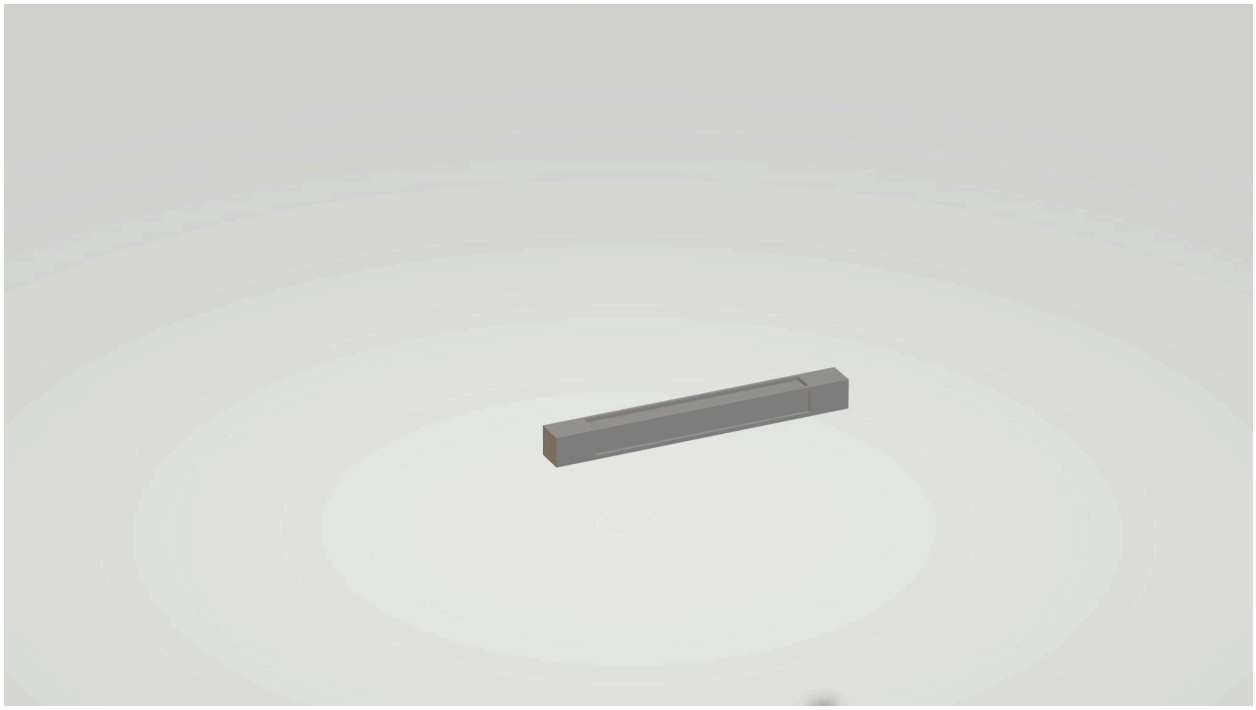


Fig. 3. 3D Printed Column

2.2.2 Software

This project made use of a Raspberry Pi 3, which hosts a Python script and receives and processes the camera feed for object detection. For object detection and visual/graphical enhancements (crosshairs, information feed, state declaration), *OpenCV* was used. Additionally, *numPy* was used for array objects, *time* was used to calculate any time deltas, and *serial* was used to communicate with the Arduino. The Arduino hosted the C++ code that controlled the wheel motors, distance sensor, and gun servos. The respective libraries for the Sabertooth motors, distance sensor, and servos were all inherently necessary.

All of the libraries used were used because they were either familiar or necessary, given the equipment available. Since OpenCV is a large and diverse library, many resources, such as the OpenCV man pages, include valuable information on certain parameters and specific types of classifiers and colour detection algorithms. Often shapes for visual elements in OpenCV are relayed in arrays, so numPy was a simple and familiar library to handle any array types.

Commands are sent between the Pi and Arduino via serial over a baudrate of 115200. These commands are one of:

- ’s’: stop

- ‘a’: autonomous roam
- ‘SLx’: Slow left x amount
- ‘SRx’: Slow right x amount
- ‘FIRE’: fire the gun

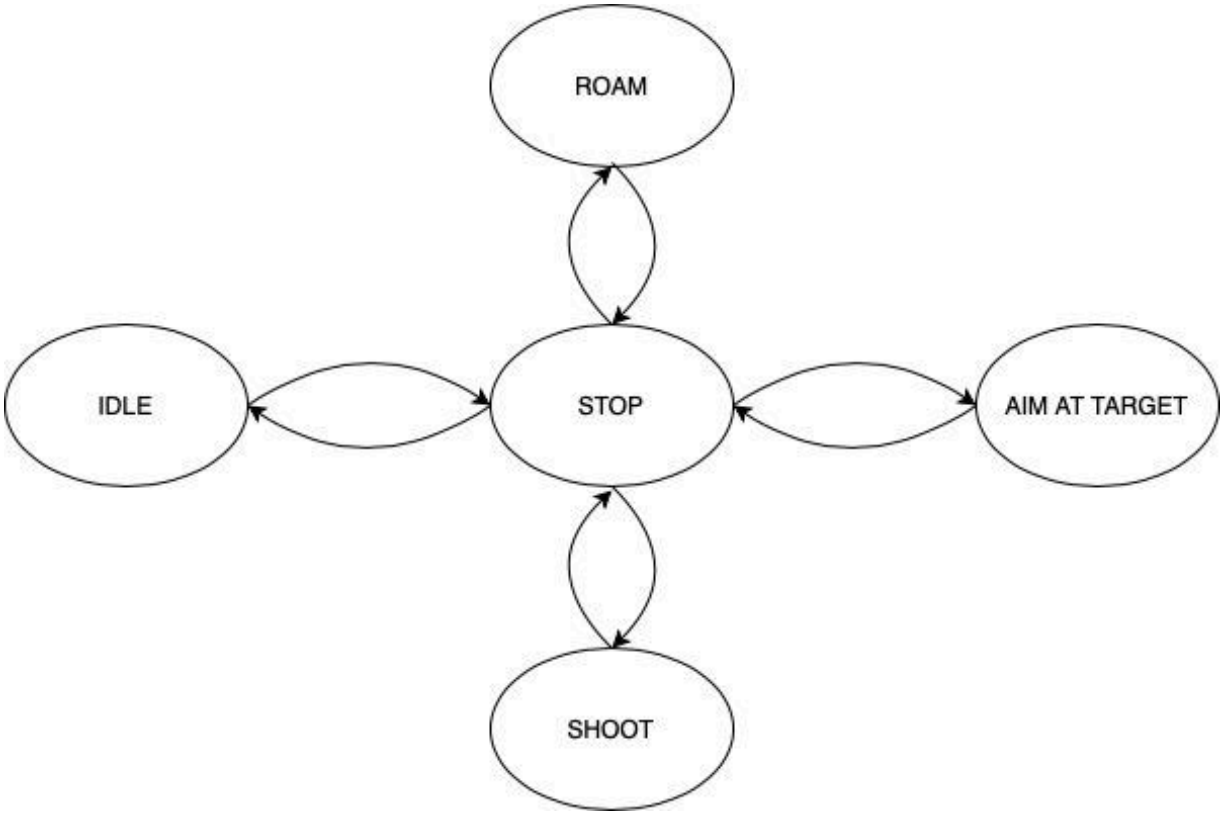


Fig. 4. State Diagram

Fig. 4 describes the states of the robot and how the states interact with each other.

Using findContours from OpenCV, red balloons are detected, and the focus is on balloons at least 14 cm in diameter, such that smaller circular red objects are not considered targets. This also accounts for the slight bit of randomness pertaining to the accuracy of the gun and the diameter of the barrel (which is much larger than the bullet). Once the target is found, the location of the balloon is calculated with respect to the origin (centre of camera frame) and the pixel lateral distance for which the robot must turn to centre the balloon is relayed to the Arduino.

The Arduino takes these commands and does as expected. The only calculating the Arduino does is it keeps checking the distance between the sensor and objects in front of it. Since the distance sensor has random spikes and is limited in its range, the robot takes the average of its readings over ten instances and takes the median as the distance reading. The threshold range for determining if an object is in the way is 120 to 330 millimetres to ensure a close distance reading is made accurate regardless of how

fast the robot is approaching an object. Fig. 5 shows the flow of the robot, decision-making junctions, and results of those decisions.

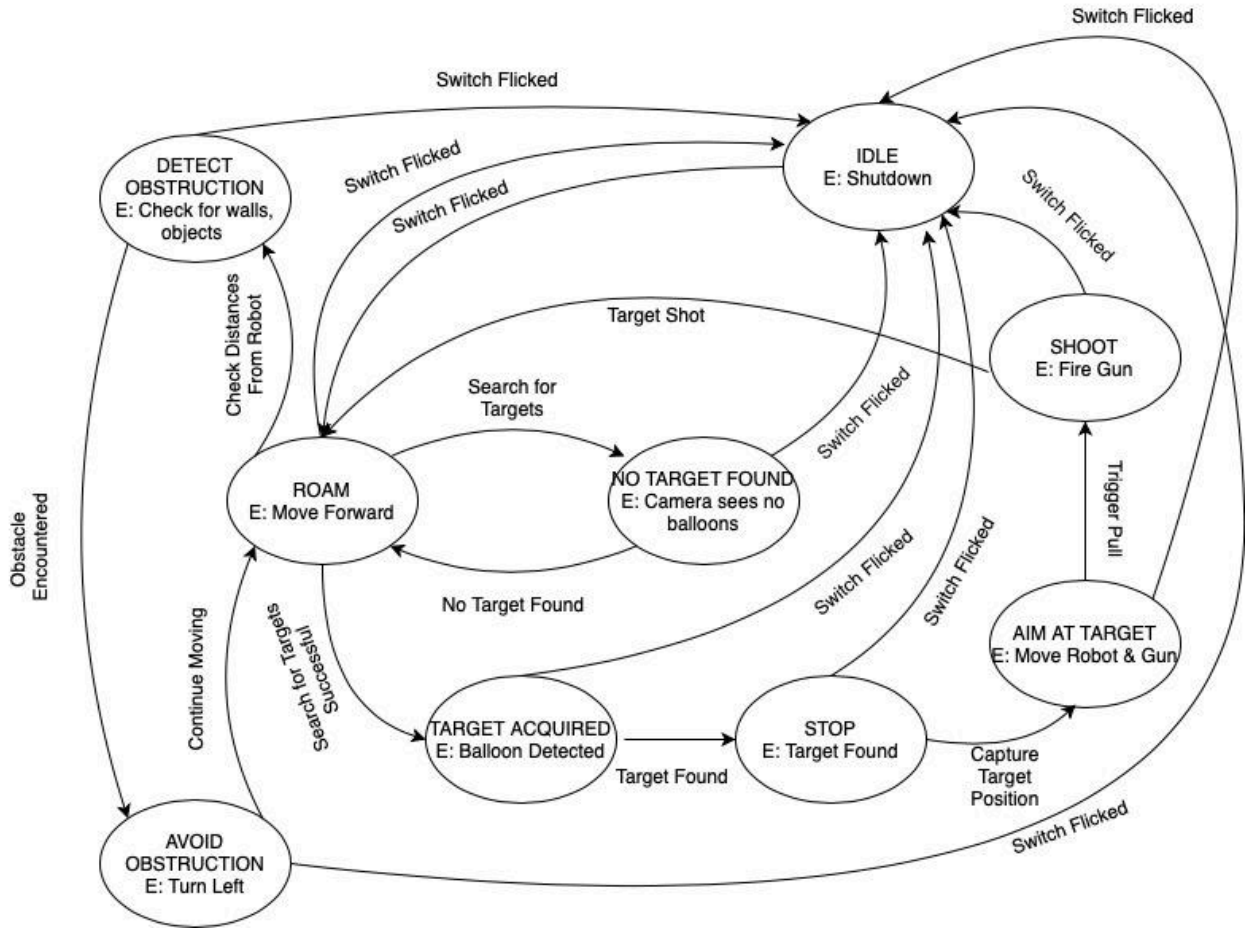


Fig. 5. Flow chart

Specific constants such as speed, pins, servo positions, etc., were all determined by trial and error. Certain constants were required to change depending on the situation (i.e., speed of the wheel motors according to the surface to prevent slipping).

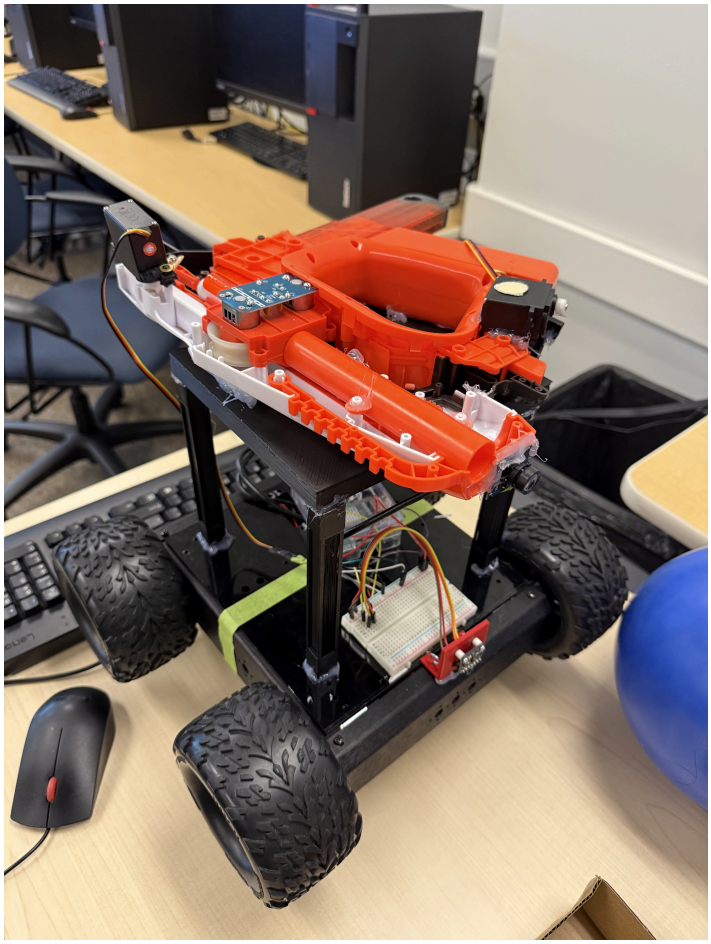


Fig. 6. Image of robot

3 Results

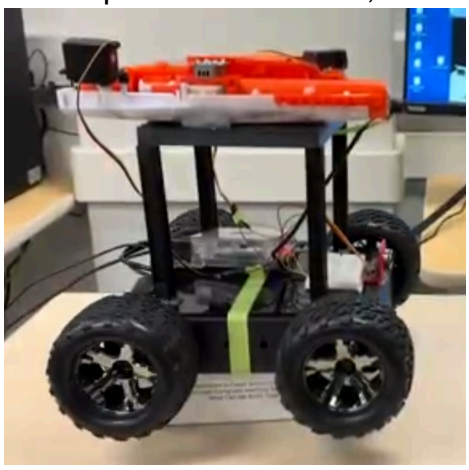
3.1 Testing

The robot was designed to autonomously identify and shoot at red balloons using computer vision and a precision shooting mechanism. Several tests were conducted to evaluate its accuracy, autonomy, and overall performance. However, due to a burnt wheel motor, all planned movement-based tests were unable to be completed on the ground and instead could only be tested with the robot propped up off the floor. Despite this limitation, preliminary results suggest that the vision and shooting systems function as expected, and theoretical analyses are provided for the untested scenarios.

TABLE 2: Tests and results

Test #	Description	Result
1	Drive forward	Theoretical Pass
2	Rotate counter-clockwise	Theoretical Pass
3	Detect obstacle within threshold (120mm)	Pass
4	Avoid obstacle	Theoretical Pass
5	Detect target by object type	Pass
6	Detect target by colour	Pass
7	Aim at red balloon only	Theoretical Pass
8	Shoot at red balloon only	Pass

Test 1 Drive forward: This test passed, but evidence was not captured before the wheel motor burnt. Image [1] shows the three remaining wheels spinning properly, all at the same speed and direction, while the robot is propped on a box.

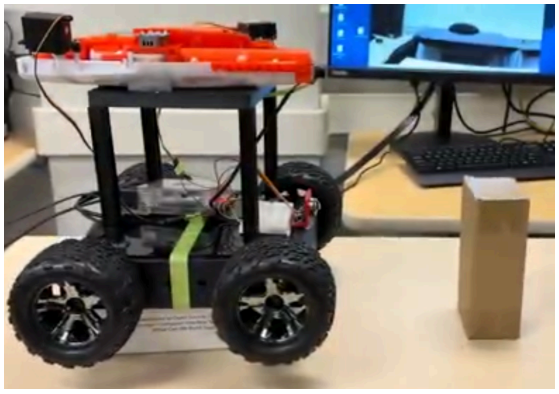


[1]

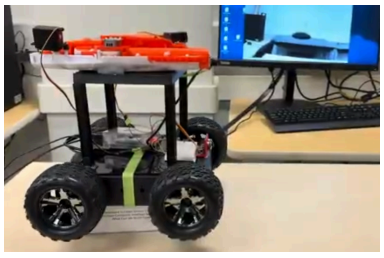
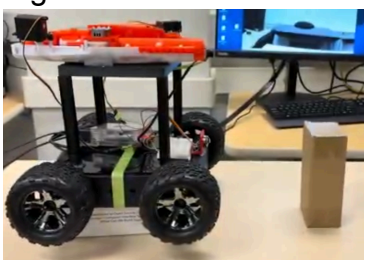
Test 2 Rotate counter-clockwise: This test passed, but evidence was not captured before the wheel motor burnt. Image [2] shows the three remaining wheels spinning properly. The two far wheels spin backward while the one close working wheel spins forward, turning the robot counterclockwise.


[2]

Test 3 Detect obstacle within threshold (120-330mm): This test passed. The test was to detect if an obstacle was directly in front of the robot. The threshold was within 120-330mm of the sensor. Images [3] show the robot driving forward while nothing is obstructing its path and image [4] shows the robot stopping once it has detected the obstacle.


[3]

[4]

Test 4 Avoid obstacle: This test passed, but evidence was not captured before the wheel motor burnt. The test was to detect if an obstacle was directly in front of the robot, and the robot would stop, turn 90 degrees counterclockwise and resume roaming. The following images show the robot driving forward while nothing is obstructing its path, stopping once it has detected the obstacle, turning the wheels to theoretically turn the robot 90 degrees, and continuing to roam forward.


[5]

[6]

[7]

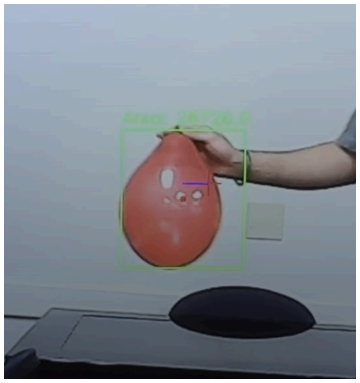
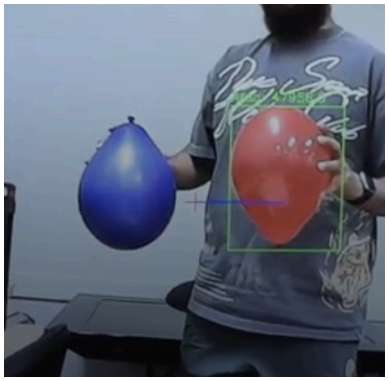
Test 5 Detect target by object type: This test passed. The test was to discern between different objects that may appear in the robot's vision. This test was performed on a human and a balloon. The robot should only identify the balloon and not the human. Image [8] shows no objects in view, image [9] shows a human in view, and image [10] shows a balloon in view. The green box around the balloon shows the working detection.


[8]

[9]

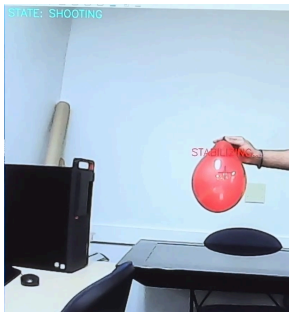
[10]

Test 6 Detect target by colour: This test passed. The test was to discern between different colored balloons. This test was performed on a red and blue balloon. The robot should only identify the red balloon. Image [11] shows only a blue balloon in which it should not identify. Image [12] shows a red balloon in which it should identify with a green box outline. Finally, image [13] shows both balloons, with only identification on the red balloon.


[11]

[12]

[13]

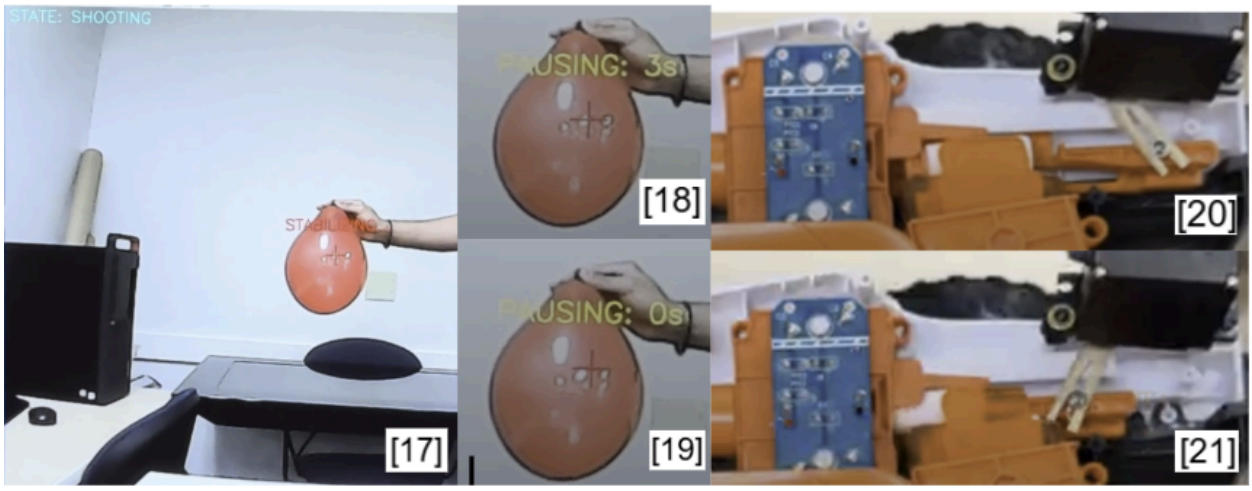
Test 7 Aim at red balloon only: This test passed. The test was to rotate the robot toward the balloon until the offset of the center of the video was aligned with the center of the target. In image [14] the top left corner of the robot video feed the state of the robot is displayed. When the robot has identified a target that is not centered, the state will be aiming. Image [15] shows the robot turning its wheels to correct its position. Image [16] shows the balloon now being in the center position and ready to shoot.

[14]

[15]


[16]

Test 8 Shoot at red balloon only: This test passed. The test was to identify the red balloon, aim towards it, and fire a single foam dart at the target using the mounted Nerf gun. After aiming at the balloon, the robot will stop all movement to stabilize and then begin to count down from three. Once it reaches zero, it will load a dart, spin the flywheels, and shoot the dart at the target. Image [17] shows the state in the top left corner is shown to be “shooting”. Images [18] and [19] show the countdown being displayed. Images [20] and [21] show the mechanism for loading the bullet. Due to the high speed of the dart, capturing photos of the dart is impossible with the robot's camera.



3.2 Conclusion

Despite encountering a critical hardware failure with one of the wheel motors, most of the planned testing met the majority of the goals. Movement tests, including driving forward and rotating counterclockwise, were successfully performed, though evidence could not be fully captured due to the motor failure. Obstacle detection, avoidance behavior, and vision-based target recognition were tested successfully, demonstrating the robot’s ability to identify and respond to objects, including distinguishing between humans and balloons, and between red and blue balloons. Finally, the robot was able to aim at and shoot red balloons with accuracy. The only two unmet goals were the full documentation of movement-based behavior, due to the hardware malfunction, and aerial targeting. However, theoretical analysis and partial visual evidence support that these features were functioning as intended prior to the wheel failure.

3.3 Future Work

While the robot successfully identified and shot at red balloons, several improvements could enhance its overall performance. The aiming algorithm could be refined for greater accuracy, and adding a vertical axis to the mounted gun would allow it to target balloons at different heights. Additionally, implementing power monitoring would help prevent unexpected failures, such as the robot unexpectedly dying due to low power. Additionally, upgrading the chassis would improve mobility, durability, and reliability, reducing the risk of mechanical failures. Finally, optimizing the roaming algorithm would enable more efficient movement and obstacle avoidance, making the robot more autonomous and adaptable in dynamic environments.

References

[1] C.-Y. Cheng, Y.-J. Chen, and S.-Y. Lin, “Design and implementation of a vision-based basketball shooting robot,” in *Proc. IEEE Int. Conf. Mechatronics (ICM)*, Taipei, Taiwan, 2005, pp. 113–117. doi: 10.1109/ICMECH.2005.1529237

[2] H. Colon, A. Horton, K. Steighner, and N. Yielding, “Autonomous Turret EEL 4914 Group 17 Fall 2010,” 2010.

[3] DFRobot FIT0701 USB Camera, “0.3 MegaPixels USB Camera for Raspberry Pi / NVIDIA Jetson Nano / UNIHIKER M10.” [Online].

[4] P. Gogoi, M. Barman, M. Deka, U. Rajkonwar, and R. Moudgollya, “Object Detection and Tracking Turret based on Cascade Classifiers and Single Shot Detectors,” in *Proc. 2020 Int. Conf. Comput. Perform. Eval. (ComPE)*, Shillong, India, 2020, pp. 792–796. doi: 10.1109/ComPE49325.2020.9200139

[5] X. Hu and R. H. Assaad, “The use of unmanned ground vehicles (mobile robots) and unmanned aerial vehicles (drones) in the civil infrastructure asset management sector: Applications, robotic platforms, sensors, and algorithms,” *Expert Syst. Appl.*, vol. 232, p. 120897, Jun. 2023.

[6] X. Liang, G. Chen, S. Zhao, and Y. Xiu, “Moving target tracking method for unmanned aerial vehicle/unmanned ground vehicle heterogeneous system based on AprilTags,” *Meas. Control*, vol. 53, no. 3–4, pp. 427–440, 2020.

[7] Q. Liu, Z. Li, S. Yuan, Y. Zhu, and X. Li, “Review on vehicle detection technology for unmanned ground vehicles,” *Sensors*, vol. 21, no. 4, p. 1354, Feb. 2021.

[8] R. K. Megalingam, V. P. Darla, and C. S. K. Nimmala, “Autonomous wall painting robot,” in *Proc. 2020 Int. Conf. Emerg. Technol. (INCET)*, Belgaum, India, 2020, pp. 1–6. doi: 10.1109/INCET49848.2020.9154020

[9] M. Ramezani, M. A. Amiri Atashgah, and A. Rezaee, “A fault-tolerant multi-agent reinforcement learning framework for unmanned aerial vehicles–unmanned ground vehicle coverage path planning,” *Drones*, vol. 8, no. 10, p. 537, Sep. 2024.

[10] Raspberry Pi 3, “Product Brief,” Apr. 2024. [Online].

[11] “VL53L0X Datasheet(PDF),” *STMicroelectronics*. https://www.alldatasheet.com/datasheet-pdf/pdf/948120/STMICROELECTRONICS/VL53L0X.html

[12] A. K. Tanyildizi, “Design, control and stabilization of a transformable wheeled fire fighting robot with a fire-extinguishing, ball-shooting turret,” *Machines*, vol. 11, no. 4, p. 492, Apr. 2023. doi: 10.3390/machines11040492

[13] A. B. Tatar, B. Taşar, and O. Yakut, “A shooting and control application of four-legged robots with a gun turret,” *Arab. J. Sci. Eng.*, Feb. 2020. Doi: 10.1007/s13369-020-04339-3

[14] C.-C. Wong, M.-F. Chou, C.-P. Hwang, C.-H. Tsai, and S.-R. Shyu, “A method for obstacle avoidance and shooting action of the robot soccer,” in *Proc. 2001 ICRA. IEEE Int. Conf. Robot. Autom.* (Cat. No.01CH37164), Seoul, Korea (South), 2001, pp. 3778–3782. doi: 10.1109/ROBOT.2001.933206

[15] Z. Zhao, Y. Zhang, L. Long, Z. Lu, and J. Shi, “Efficient and adaptive lidar–visual–inertial odometry for agricultural unmanned ground vehicle,” *Int. J. Adv. Robot. Syst.*, vol. 19, nos. 2, pp. 1–15, Mar.–Apr. 2022.

[16] ”Sabertooth 2x12 RC User’s Guide” https://cdn.robotshop.com/media/D/Dim/RB-Dim-43/pdf/sabertooth-dual-12a-regenerative-motor-driver-quickstart.pdf

Appendices

Appendix - A: Code

Arduino Code:

```
#include <Wire.h>
#include <Servo.h>
#include "Adafruit_VL53L0X.h"

// Distance sensor setup
Adafruit_VL53L0X lox = Adafruit_VL53L0X();
const int numSamples = 10;
int distances[numSamples];

// Drive motors
Servo motorLeft;
Servo motorRight;

// Gun servos
Servo reloadServo;
Servo shootServo;

// Motor constants
const int STOP = 1500;
const int FULL_FORWARD = 1100;
const int FULL_REVERSE = 1900;
const int SLOW_FORWARD = 1400;
const int SLOW_TURN_SPEED = 100;
const int TURN_DURATION = 600;

// Gun servo constants
const int servoReloadPin = 6;
const int servoShootPin = 5;
const int reloadRetractedPos = 190;
const int reloadExtendedPos = 60;
const int shootIdlePos = 20;
const int shootActivePos = 50;

// Control variables
unsigned long lastCommandTime = 0;
const unsigned long commandTimeout = 2000;
String input = "";
bool autonomousMode = true;
bool shootingInProgress = false;
```

```
void setup() {
  Serial.begin(115200);

  // Initialize drive motors
  motorLeft.attach(9);
  motorRight.attach(11);
  motorLeft.writeMicroseconds(STOP);
  motorRight.writeMicroseconds(STOP);

  // Gun servos
  shootServo.write(shootIdlePos);

  // Distance sensor
  Wire.begin();
  if (!lox.begin()) {
    Serial.println("Failed to boot VL53L0X");
    while (1);
  }
  lox.startRangeContinuous();

  Serial.println("Robot initialized and ready");
}

void loop() {
  // Check for serial commands
  while (Serial.available()) {
    char c = Serial.read();
    if (c == '\n') {
      processCommand(input);
      input = "";
      lastCommandTime = millis();
    } else {
      input += c;
    }
  }

  // Run autonomous mode if active and we haven't received commands recently
  if (autonomousMode && millis() - lastCommandTime > commandTimeout &&
!shootingInProgress) {
    autonomousRoam();
  }
}

void autonomousRoam() {
  int medianDistance = getMedianDistance();
```

```
  if (isClose(medianDistance)) {
    Serial.println("Wall detected, turning left.");
    turnLeft90();
  } else {
    motorLeft.writeMicroseconds(SLOW_FORWARD);
    motorRight.writeMicroseconds(SLOW_FORWARD);
  }
  delay(50);
}

int getMedianDistance() {
  for (int i = 0; i < numSamples; i++) {
    while (!lox.isRangeComplete()) {
      delay(1);
    }
    distances[i] = lox.readRange();
  }
  sortArray(distances, numSamples);
  return distances[numSamples / 2];
}

bool isClose(int distance) {
  return (distance > 120 && distance < 330);
}

void turnLeft90() {
  motorLeft.writeMicroseconds(STOP - SLOW_TURN_SPEED);
  motorRight.writeMicroseconds(STOP + SLOW_TURN_SPEED);
  delay(TURN_DURATION);
  motorLeft.writeMicroseconds(STOP);
  motorRight.writeMicroseconds(STOP);
}

void processCommand(String cmd) {
  cmd.trim();

  if (cmd == "a") {
    Serial.println("Autonomous mode enabled.");
    autonomousMode = true;
  }
  else if (cmd == "s") {
    Serial.println("Stopping robot.");
    motorLeft.writeMicroseconds(STOP);
    motorRight.writeMicroseconds(STOP);
    autonomousMode = false;
```

```
  }
  else if (cmd == "FIRE") {
    Serial.println("Firing dart gun!");
    autonomousMode = false;  // Disable autonomous mode during shooting
    fireGun();
  }
  else if (cmd.startsWith("SL") || cmd.startsWith("SR")) {
    int turnTime = cmd.substring(2).toInt();
    if (turnTime > 0) {
      if (cmd.startsWith("SL")) {
        Serial.println("Slow left turn.");
        motorLeft.writeMicroseconds(STOP - SLOW_TURN_SPEED);
        motorRight.writeMicroseconds(STOP + SLOW_TURN_SPEED);
      } else {
        Serial.println("Slow right turn.");
        motorLeft.writeMicroseconds(STOP + SLOW_TURN_SPEED);
        motorRight.writeMicroseconds(STOP - SLOW_TURN_SPEED);
      }
      delay(turnTime);
      motorLeft.writeMicroseconds(STOP);
      motorRight.writeMicroseconds(STOP);
    }
  }
}

void fireGun() {
  // Set flag to true
  shootingInProgress = true;

  // Stop the robot first
  motorLeft.writeMicroseconds(STOP);
  motorRight.writeMicroseconds(STOP);

  // Attach servos
  reloadServo.attach(servoReloadPin);
  shootServo.attach(servoShootPin);

  // Spin motors up
  for (int pos = shootIdlePos; pos <= shootActivePos; pos += 2) {
    shootServo.write(pos);
    delay(10);
  }

  // Push bullet position
  for (int pos = reloadRetractedPos; pos >= reloadExtendedPos; pos -= 5) {
    reloadServo.write(pos);
```

```
    delay(20);
  }

  // Return to retracted position
  for (int pos = reloadExtendedPos; pos <= reloadRetractedPos; pos += 1) {
    reloadServo.write(pos);
    delay(15);
  }

  // Stop spinning motors
  for (int pos = shootActivePos; pos >= shootIdlePos; pos -= 2) {
    shootServo.write(pos);
    delay(15);
  }

  // Final positioning
  shootServo.write(30);
  delay(200);
  reloadServo.write(160); // Move back to retracted position for next bullet
  delay(700);

  // Detach servos
  reloadServo.detach();
  shootServo.detach();

  // Clear the shooting flag
  shootingInProgress = false;

  Serial.println("Firing sequence completed");
}

void sortArray(int arr[], int size) {
  for (int i = 0; i < size - 1; i++) {
    for (int j = 0; j < size - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
}
```

Raspberry Pi Code

```
import cv2
import numpy as np
import serial
import time

arduino = serial.Serial('/dev/ttyACM0', 115200)
time.sleep(2)

cap = cv2.VideoCapture(0)

# Camera resolution is 1280x960
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 960)

# Robot states
STATE_ROAMING = 0
STATE_AIMING = 1
STATE_SHOOTING = 2
STATE_PAUSING = 3
current_state = STATE_ROAMING  # since ROAMING is set to True on startup on
Ardunio Code
last_state_change = time.time()


def send(cmd):
    """
    Purpose: Send the command to Arduino.
    Input: cmd <type: str> which reflects the serial command sent to Arduino
    Output: None
    """
    arduino.write((cmd + '\n').encode())
    print(f"Sent: {cmd}")
    time.sleep(0.1)  # Adding delays throughout help adjust for any lag between
communication


def detect_red_balloon(frame):
    """
    Purpose: Detect red balloons and return the center coordinate and area if all valid.
    Input: frame <type: int> from camera
    Output: x <int>, y <int>, z <int> coordinates of balloon (x,y) and square area of
balloon (z).
    """
```

```
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # So the red color range is on both ends of the HSV
    lower_red1 = np.array([0, 100, 50])
    upper_red1 = np.array([10, 255, 255])
    lower_red2 = np.array([170, 100, 50])
    upper_red2 = np.array([180, 255, 255])

    mask1 = cv2.inRange(hsv, lower_red1, upper_red1)
    mask2 = cv2.inRange(hsv, lower_red2, upper_red2)
    mask = cv2.bitwise_or(mask1, mask2)

    # This should reduce some noise in detection, I'll test for difference.
    kernel = np.ones((5, 5), np.uint8)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    if contours:
        largest_contour = max(contours, key=cv2.contourArea)  # i.e., biggest red circle
        area = cv2.contourArea(largest_contour)

        if area < 3000:  # if red circle is small: ignore
            return None, None

        x, y, w, h = cv2.boundingRect(largest_contour)  # else: save centre of bounding
rectangle around balloon
        center_x = x + w // 2

        # //VISUALS//
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
        cv2.circle(frame, (center_x, y + h // 2), 5, (0, 0, 255), -1)
        cv2.putText(frame, f"Area: {area}", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX,
0.9, (0, 255, 0), 2)

        return center_x, area

    return None, None  # No valid balloon detected


def shoot():
    """
    Purpose: send a shoot command to Arduino.
    Input: None
    Output: None
```

```
    """
    send("FIRE")
    print("FIRING!")


if __name__ == "__main__":
    try:
        while True:
            ret, frame = cap.read()
            if not ret:
                print("cap.read() failed in main loop.")
                break

            # draws a crosshair
            h, w = frame.shape[:2]
            center_x, center_y = w // 2, h // 2
            cv2.line(frame, (center_x - 20, center_y), (center_x + 20, center_y), (0, 0, 255),
2)
            cv2.line(frame, (center_x, center_y - 20), (center_x, center_y + 20), (0, 0, 255),
2)

            # might not need
            current_time = time.time()

            if current_state == STATE_ROAMING:
                # Look for balloons while roaming
                balloon_x, balloon_area = detect_red_balloon(frame)

                if balloon_x is not None:
                    # Balloon detected! Stop and switch to aiming state
                    send('s')  # 's' means stop
                    current_state = STATE_AIMING
                    last_state_change = current_time
                    print("Balloon detected! Stopping to aim...")
                else:
                    # No balloon, ensure we're in roaming mode
                    if current_time - last_state_change > 5:  # Send roam command
periodically
                        send('a')  # btw 'a' means roam, but that's short for automatic_roam()
from Arduino
                        last_state_change = current_time

            elif current_state == STATE_AIMING:
                # Fine-tune aim at the balloon SL or SR for slow-left or slow-right
                balloon_x, balloon_area = detect_red_balloon(frame)
```

```
            if balloon_x is None:
                # if balloon goes out of frame then we'll go back to roaming
                current_state = STATE_ROAMING
                send('a')
                last_state_change = current_time
                print("Lost balloon, resuming roam...")
            else:
                # Offset from center is the vector to find distance needed to adjust SL or
SR
                offset = balloon_x - center_x

                # Draw horizontal targeting line
                cv2.line(frame, (center_x, center_y), (balloon_x, center_y), (255, 0, 0), 2)
                cv2.putText(frame, f"Offset: {offset}", (50, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)

                # If we've been aiming for at least 1 second, start turning if needed
                if current_time - last_state_change > 1:
                    # if "close enough" to centre, then shoot
                    if abs(offset) < 30:  # works best in testing, might need to adjust once
mounted fully
                        current_state = STATE_SHOOTING
                        last_state_change = current_time
                        print("Target locked! Preparing to shoot...")  # crosshair is lined up,
now shoot
                    else:
                        # turn duration
                        turn_factor = min(abs(offset) / 50, 3)  # scaled
                        turn_time = int(turn_factor * 100)  # ^

                        if offset < 0:
                            send(f"SL{turn_time}")  # Slow left
                            print(f"Turning left for {turn_time}ms")
                        else:
                            send(f"SR{turn_time}")  # Slow right
                            print(f"Turning right for {turn_time}ms")

                        last_state_change = current_time  # timer resets after each
adjustment

        elif current_state == STATE_SHOOTING:
            # We're in position to shoot
            if current_time - last_state_change < 2:  # Wait 2 seconds to stabilize from
wiggle or jerk
                cv2.putText(frame, "STABILIZING...", (center_x - 100, center_y - 50),
                            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
```

```
            else:  # BANG
                shoot()
                current_state = STATE_PAUSING
                last_state_change = current_time
                print("Shot fired! Pausing...")

        elif current_state == STATE_PAUSING:
            # Delay after shot is fired
            cv2.putText(frame, f"PAUSING: {3 - int(current_time - last_state_change)}s",
                        (center_x - 100, center_y - 50), cv2.FONT_HERSHEY_SIMPLEX, 1,
(0, 255, 255), 2)

            if current_time - last_state_change > 3:  # Pause for 3 seconds
                current_state = STATE_ROAMING
                send('a')  # Resume roaming
                last_state_change = current_time
                print("Resuming roam...")

        states = ["ROAMING", "AIMING", "SHOOTING", "PAUSING"]
        cv2.putText(frame, f"STATE: {states[current_state]}", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 0), 2)

        cv2.imshow("Balloon Detection", frame)

        key = cv2.waitKey(1) & 0xFF
        if key == ord('q'):
            break
        elif key == ord('s'):
            send('s')
        elif key == ord('a'):
            send('a')

    finally:  # stop turret and end
        send('s')
        cap.release()
        cv2.destroyAllWindows()
        arduino.close()
        print("Program terminated safely")
```