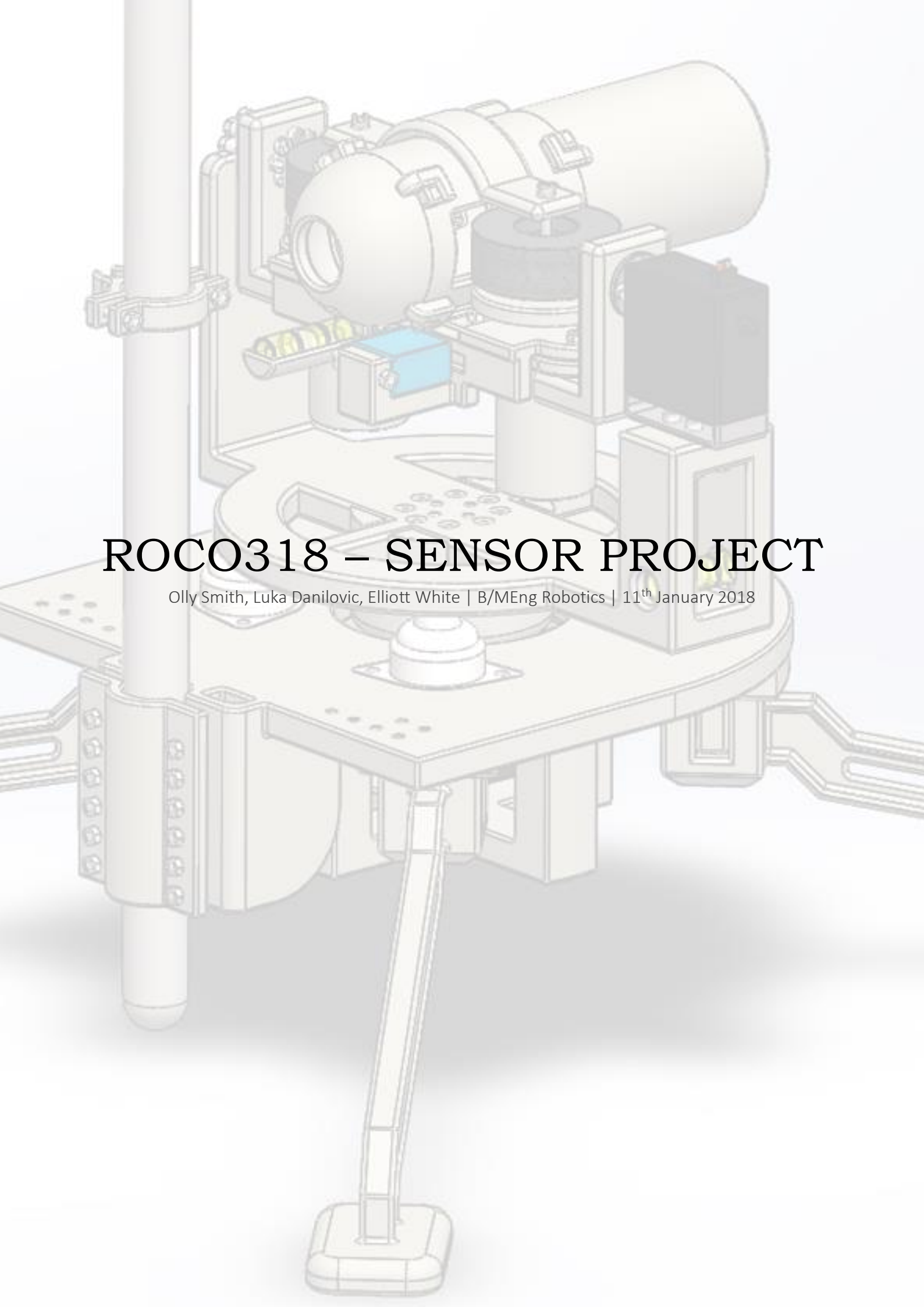


ROCO318 – SENSOR PROJECT

Olly Smith, Luka Danilovic, Elliott White | B/MEng Robotics | 11th January 2018

Contents

1. Brief

1.1 Introduction

As part of the ROCO318 Mobile and Humanoid Robots module, Séverin Lemaignan set a group task of utilising a sensor for a novel application, with the goal of developing a firm understanding of the sensors merits and drawbacks, evaluating its use in our group project.

Our group (Olly Smith, Luka Danilovic, Elliott White) chose to build a small robot turret; capable of tracking a target, firing a ping pong ball at the target, and the adjusting aim should the ball miss.

This report details the design and construction of our turret, as well as analysis and evaluation of our chosen sensor, the PlayStation Eye digital webcam.

1.2 Project Brief: Ping pong ball turret

Our group chose to make a small autonomous turret, that would track and fire ping pong balls at a target.

The turret itself would comprise of a stationary platform, and a firing mechanism, controlled by two servo motors (for X & Y rotation about the centre of the mechanism), and using two dc motors to fire a ping pong ball, pushed between two spinning wheels thanks to a third small servo. The camera would be mounted in a fixed position above the firing mechanism, with a clear view of the operating environment and unobstructed by the turret below.

The turret would be controlled by a microcontroller, and the sensor would be plugged into a laptop, which would perform the necessary machine vision and calculations, before sending commands to the microcontroller, in a master-slave setup.
When operating, the program would look for the target, subsequently tracking the position and estimating the distance. The program would then perform trajectory calculation, and send the X & Y servo positions to the microcontroller, which would set the servos to the positions specified.

When the command to fire is sent (a key press) the turret would fire a ping pong ball at the target, and the laptop/camera would track to see if the ball hit the target, or else how far off the ball was, before adjusting the turret angles accordingly.

1.3 Sensor Brief: PlayStation Eye

Figure 1. The PlayStation Eye camera.

Description

The PlayStation Eye is a digital camera and microphone device designed for the PlayStation 3 gaming console. The camera features the capability of capturing video at high frame rates. The camera was chosen specifically for its high frame rate, more about this can be found under camera selection.

Features and use

Looking at the instruction manual[1] contains lots of useful information about the camera, notably:

- Recommended range (for use with video games) 1.5 – 2.0m
- Manual FOV (field of view) selection, a manually operated lens angle selector with two settings, 56 close up, or 75 degree wide shot. These options are marked on the camera as a red and blue dot, respectively.
- USB 2.0, type A connection
- Frame rate of 60 fps (frames per second) at 640 x 480, or 120 fps at 320 x 240, using uncompressed video format

Omnivision OV7220 CMOS VGA Sensor

Through investigation it appears that the chip used in the sensor is the Omnivison OV7221, the datasheet for which is available online [2].

No official drivers are available for the Eye, however there is a windows driver available [3], and the camera works directly with Ubuntu 16.04. Additionally, there is a Qt VL42 test utility program (figure 2), which allows for “easy experimenting with video4linux devices”. This proved very useful for understanding what the camera can do.

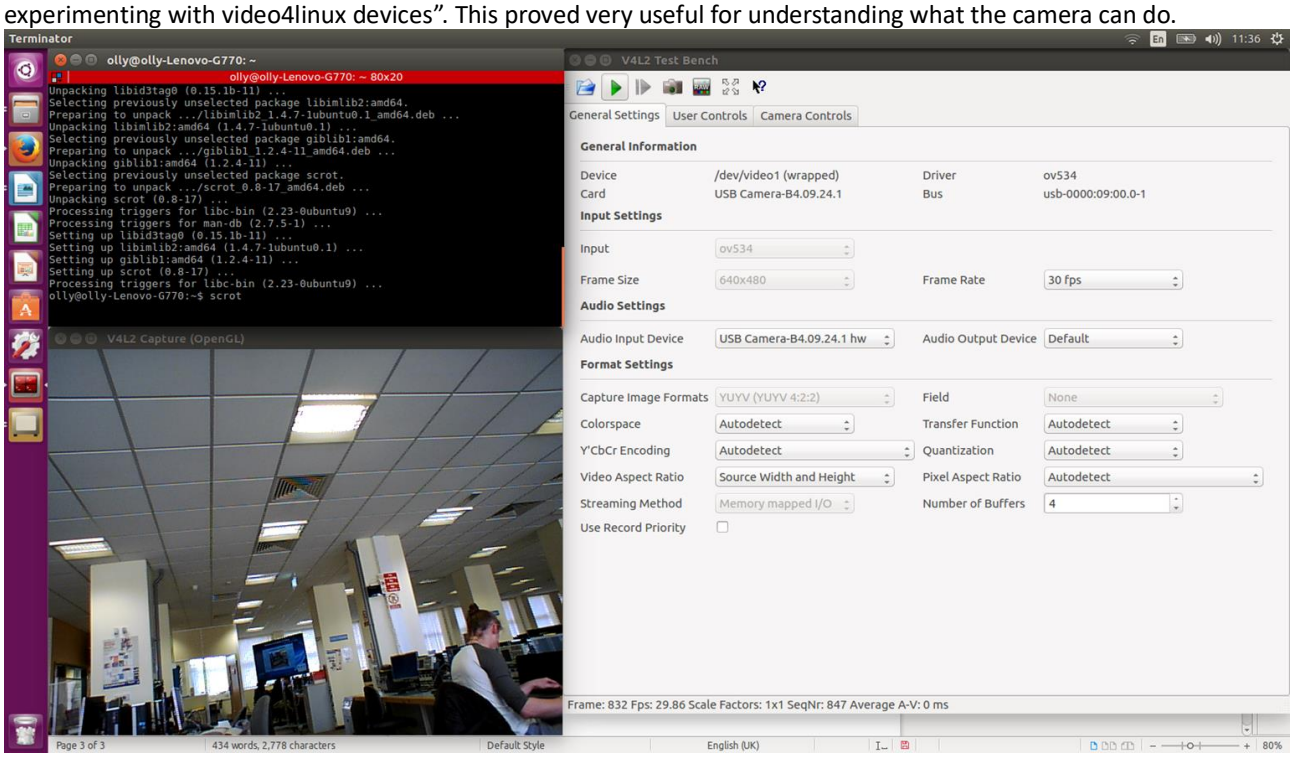


Figure 2. The Qt VL42 test bench, with video output.

The camera handles a wide variety of lighting conditions far better than comparative webcams. Figure 2 illustrates this, comparing the EyeToy, and the onboard webcam of my laptop.

Product History

The product was developed for the PlayStation 3 gaming console, as a successor to the PlayStation EyeToy, a PlayStation 2 accessory. Its use was for “players to interact with games using motion and colour detection as well as sound through its built in microphone array.” It was developed as part of PlayStation Move, a motion game controller of similar like to the Nintendo Wii remotes. However, whereas the Wii remotes used infrared to track the controller positions, the Move used the PlayStation Eye to track coloured balls that were a part of the controllers.

Figure 3. PlayStation Move controller (left), Nintendo Wii Remote (right).

However, while the original EyeToy and Wii were both commercial successes (selling 10.5m worldwide & 8.5m in the US, respectively), the Move controllers, and thus the PlayStation Eye, failed to meet their sales expectations (selling 15m globally as of 2012), and the Sony UK VP acknowledged that the device did not meet expectations[4].

The result of the Moves commercial failure, and thus the PlayStation Eye, is that units can now be bought for as little as 75p [5], which for a camera capable of 120fps is incredible. As such, the camera could well be a great prototyping and development tool for robotics, whilst supplies of the discontinued product are still available.

Applications & Limitations

The camera offers the potential for a great value for money high speed camera. Similar cameras such as the Pi Camera V2[6], or the Matrix Vision mvBlueFox[7] have a max framerate of 90fps, and are more expensive. However, due to the PS3 EyeToy being a discontinued product, the camera would not be fit to commercial products.

The PS3 EyeToy camera would be well suited as a prototyping tool, for high speed machine vision applications, such as robotics, automation or research. Were a prototype then further developed into a commercial product, a replacement camera would be sought for production.

2. Design Process

This chapter details the design, manufacture, and testing of the turret.

2.1 Conception

When discussing ideas for the project we wanted something that would be fun to build, suitably challenging for us to develop our understanding of the sensor, and crucially, something achievable within the time available.

After discussing a number of ideas internally and with Severin and Emmanuel, we settled on the ping pong ball turret. The idea allowed us to divide responsibility well, as Luka had experience with CAD, Elliott with embedded systems programming (coming off a year in industry) and myself with computer vision (having experimented with OpenCV in preparation for my third-year project). Additionally, we felt the idea offered a good combination of hardware, software & mechanical design, and that we had the capability to deliver the project in five weeks.

2.2 ROS

Initially, we wanted to use ROS (the Robot Operating System)[8]. From Wikipedia:

“ROS is a Robotics middleware (i.e. collection of software frameworks for robot software development). Even though ROS is not an operating system, it provides services designed for heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages.”

ROS features a wide variety of topics (named busses over which nodes exchange messages), that can be easily linked together (in a publisher/subscriber relationship) to perform complex tasks. Examples of topics range from graphical interfaces (rviz) and keyboard inputs (keyboard), to RGBd drivers for the Microsoft Kinect sensor (kinect_node), robot path planning (global_planner), or virtual simulation (gazebo).

In theory ROS can be used to easily connect topics to create a framework for advanced robotics. Robots such as Baxter[9] and Robonaut[10] For our turret, the intention was to use existing topics to utilise of camera, model our robot and target coordinates in a virtual environment, and perform trajectory calculations for the ball.

However, setting up ROS on a laptop proved to be harder than anticipated. The installation itself was painless, but the cameras we attempted to interface did not properly display depth information. These problems persisted on the university computers.

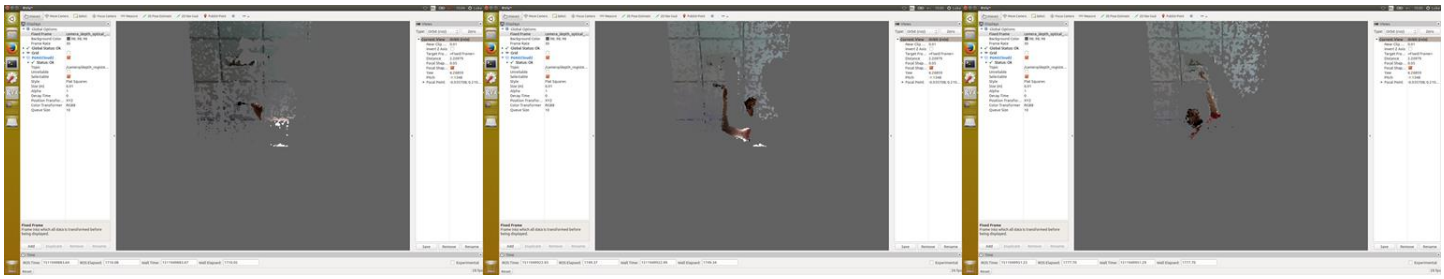


Figure 4. Screenshots of the ping pong ball using rviz depth cloud at 50,150, and 300cm.

As demonstrated in figure 4, the ping pong ball is only just visible at 50cm, and is indistinguishable at 3 meters. From this it can be concluded that the RealSense cameras are not capable of tracking a small, fast moving object, over a distance.

Communicating from a laptop to a microcontroller was not too difficult, but was still confusing. After following the tutorials on the ROS wiki, we were able to get a couple of simple subscriber programs to work on the NUCLEO-F429ZI to move a servo and toggle LEDs. This was done by sending integers or characters to the microcontroller from a publisher, and then interpreting the message. Even though this was relatively simple, we were unsure how to process camera information and then send servo positions to the microcontroller.

We persevered with ROS for two weeks, at the end of which we had basic control of the RGBd camera, and could visualize a point cloud of the depth sensor. At this point we saw that the RGBd cameras would not be fit for purpose (see camera selection), and decided to abandon ROS. We recognise the value of ROS as a robotics platform, but felt that it overcomplicated our project in respect to the task, and that we could achieve results faster without it.

2.3 Camera Selection

A number of sensors were provided for use by the faculty. We quickly narrowed these down to a short list, and evaluated them in respect to our project.

Microsoft Kinect V1
The Kinect features an RGBd sensor, with a range of X and 2d image of (RES & FPS). While these may have been suitable, the Kinect required using a SDK (software developer kit) the was exclusive to the windows operating system. For this reason, we ruled out the Kinect sensor, as we wanted to work from a Linux platform for our programming.

Intel RealSense R200
As mentioned in figure 4, the RealSense camera is not able to properly track the ball using the depth aspect of the camera.

Intel RealSense creative vf800
Like the R200, the vf800 depth sensor is not suited for tracking a small fast-moving object due to lack of detail provided by the depth cloud.

Framerate problem
A common problem all our RGBd cameras share is a slow framerate which would prevent us from accurate tracking in the period when the ball would hit the target.

PS3 EyeToy
After ruling out RGBd cameras we needed to find a camera that would easily interface with the computer, work well in a variety of lighting conditions, and a high enough framerate to track the ball in flight.

We quickly discovered the PlayStation 3 EyeToy, a usb2.0 webcam capable of 60fps at 640 x 480, or 120 fps at 320 x 240. The camera output uncompressed video, was designed for machine vision purposes in mind, and was available from CeX for 50p.

More about the camera can be found in the sensor brief.

2.4 Code design

The code for the turret can be split into two aspects: The OpenCV machine vision code for the target acquisition & trajectory planning, and the microcontroller code for controlling the turret's servo's.

hsv_slide.py
Used to calibrate target and ball HSV settings for the operating environment. The program contains sliders which allows for experimentation of the hue, saturation, and value settings, to find the optimal values for the object in question. Once the values have been found, they are manually set in the colour ranges of the main program.

Target acquisition and trajectory planning
The machine vision code works in four steps:

Initially, the code masks the frame for the HSV ranges, for both the ball and the target. Next, these masks are used by the draw_rect and draw_circle functions. The draw_circle function uses moments to estimate the centre x & y coordinates of the ball, and applies a circle and point for visualisation purposes.

The draw_rect function is more complicated. It estimates the centre point coordinates from the four corner points of the square.

The distance estimation of the target was relatively simple. We needed a calibration image of the target that was a known distance from the camera, and know width. From this we would be able to calculate the focal length of the camera. Once this was known, the calculation could be reversed, and we would be able to calculate the distance of the target from the camera.

$$\text{Focal Length} = (\text{Perceived Width} * \text{Known Distance}) / \text{Known Width}$$

The perceived width of the target is the width in pixels, which can be found by drawing a boundary around the target and then extracting the width information. Once the calibration was done, we can call a separate function to calculate the distance of the target from the camera.

$$\text{Distance from camera} = (\text{Known Width} * \text{Focal Length}) / \text{Perceived Width}$$

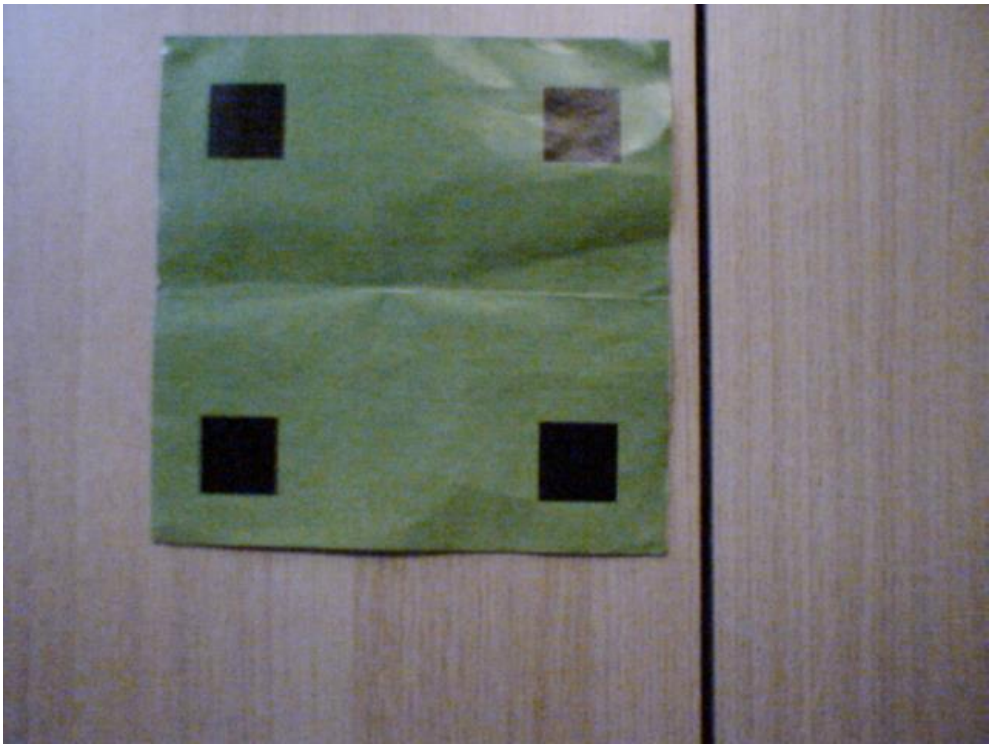


Figure 5. Initial target distance calibration image (30cm from camera)

To determine if the initial calibration image was suitable, we used multiple images of known increasing distances and run the algorithm to see what the calculated distance was.

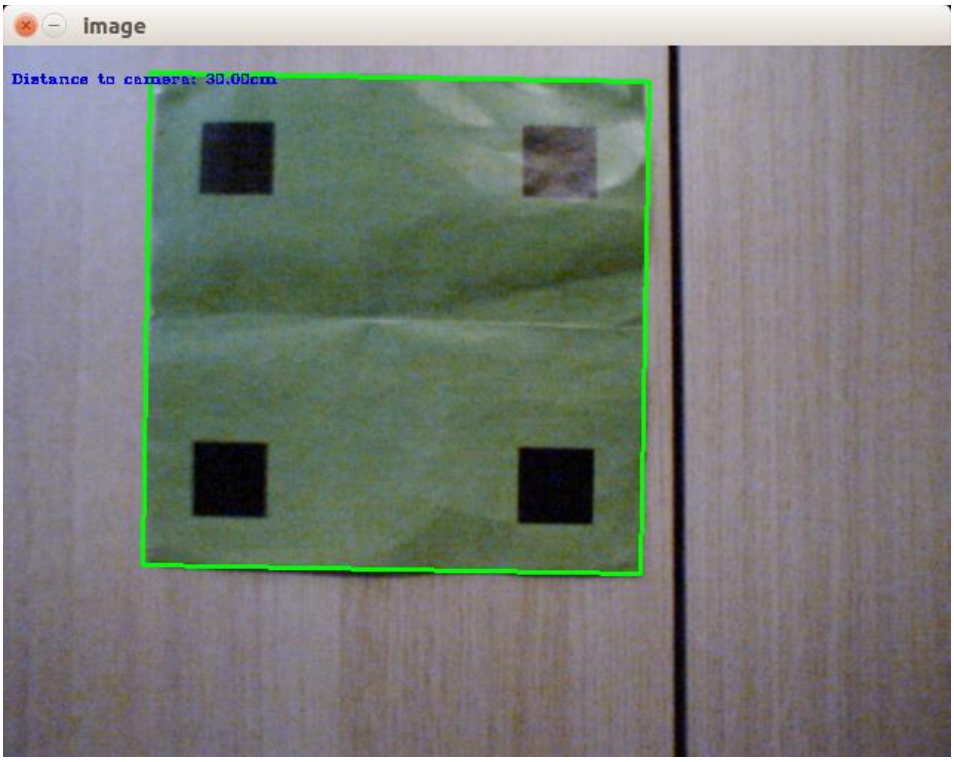


Figure 6. Initial target distance calibration image with boundary

Figure 7. 60cm from camera image. Calculated distance is 58.62cm

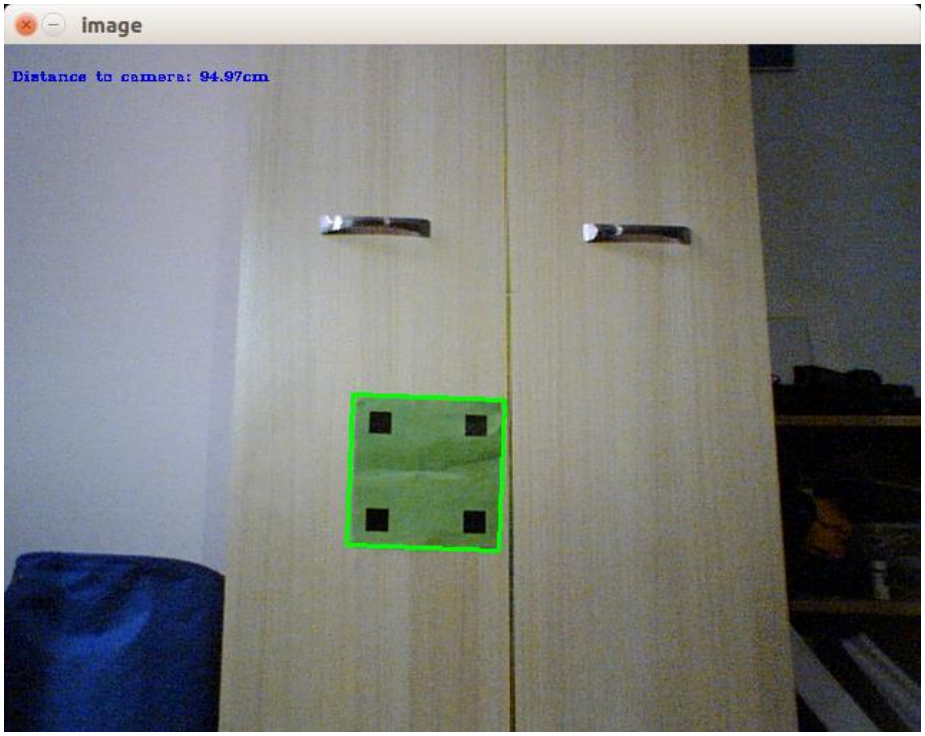


Figure 8. 90cm from camera image. Calculated distance is 94.97cm

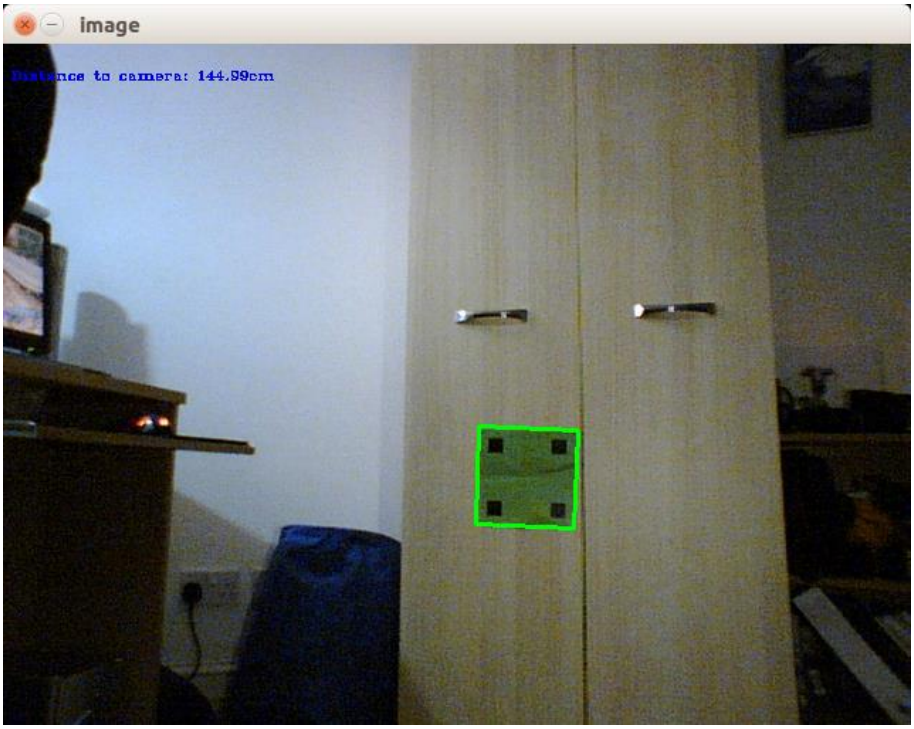


Figure 9. 150cm from camera image. Calculated distance is 144.99cm

The calculated distances of the target were very close to the actual distances, so we decided the initial calibration image was suitable. Variations of +/- 5cm would not make a difference as to whether the ball would hit the target.

To calculate the trajectory of the ball, we used the formula found at https://en.wikipedia.org/wiki/Projectile_motion (Angle required to hit coordinate (*x*,*y*)). Using this, we would be able to calculate the launch angle needed for the ball to hit the target at a known distance and at a known elevation of the turret. We also used a predetermined initial velocity of the ball as we had not yet calculated this.

$$\theta = \arctan\left(\frac{v^2 \pm \sqrt{v^4 - g(gx^2 + 2yv^2)}}{gx}\right)$$

Figure X. Vertical launch angle calculation, where x is a known height of the turret above the target, and y is the known distance of the target. v is the initial velocity, and g is acceleration due to gravity

Once this was calculated, we would then send this down the serial port to the NUCLEO board.

The time of flight could also be calculated:

$$t = \frac{2v_0 \sin(\theta)}{g}$$

After having calculated this, we would be able to check if the ball had hit the target after the calculated time. We start a timer, and after the calculated time has passed since launching the ball, check to see if the contour of the ball is within the contour of the target. If it is, the ball has hit the target, if not, then it has missed.

2.5 Microcontroller turret control

To control the servos, we used simple serial communication with the NUCLEO-F429ZI. This was done using the serial library of Python. We located the centre position of the target from the camera, and then used trigonometry to determine how much the turret should turn. If we knew the distance of the target, and how far away from the centre of the image it was, then calculating the turning angle wouldn't be difficult. We also know the perceived width and actual width of the target, so could work out how many centimetres each pixel in the image represented. Once both the turning angle and vertical angle were calculated, they would be sent down to the microcontroller every 60 frames. After the first 60 frames, the algorithm would send the turning angle (x), and after the next 60 frames, the launch angle would be sent (y). The number of frames between sending information can be changed, but we found this number to be best as we did not need the turret to constantly be adjusting itself as the target was not moving.
The algorithm would send two strings at a time to the microcontroller. First, which servo to be controlled (x or y), then the angle to move that servo. This method was chosen, as oppose to sending the servo choice and angle in one string, because it would be easier on the microcontroller coding side to read separate strings than try to extract integers from a string that contains both text and numbers.

The microcontroller starts by initialising the servos and setting them to home positions. It then waits for serial data to be sent from the PC. Once data has been received, it compares the string received to "xServo" or "yServo" to determine which servo it will be controlling. It will then wait for another string to be received, extract the angle and then convert it into suitable servo control code. After this, the input buffer gets reset and it will go back to waiting for the servo selection string. The microcontroller also sends some debug information back to the PC, so if a serial port monitor is opened, we can see what the microcontroller has received and see if it is the same as what was sent.

All code can be found in the appendix.

2.6 CAD Design and Manufacture (chronological description)

After some hand drawn sketches and agreeing on the requirements of the design we decided that it would be best if the turret has a camera that sits high above the barrel to provide a clear Arial view of the target as well as separately controlled elevation, azimuth & launch mechanisms.

The rotational servos are to be positioned in a way that rotating the turret would result in the ball, at the point of fire, to be rotated around its core and be stationary translation wise.

The launch mechanism would sit nested into the rotation mechanism and consist of two horizontal fly-wheels that would spin up and then grip & fire the ball.
For stability the whole assembly would sit on a flat, wide base.

Aware of the time constraints and having consulted with various manufacturing departments, we agreed that the best material/process combination for this project would be laser cutting plywood of different thickness with only the most critical parts being 3D printed.

With this in mind, most parts were ordered, and the first prototype was designed in SolidWorks to fit together like a jigsaw.

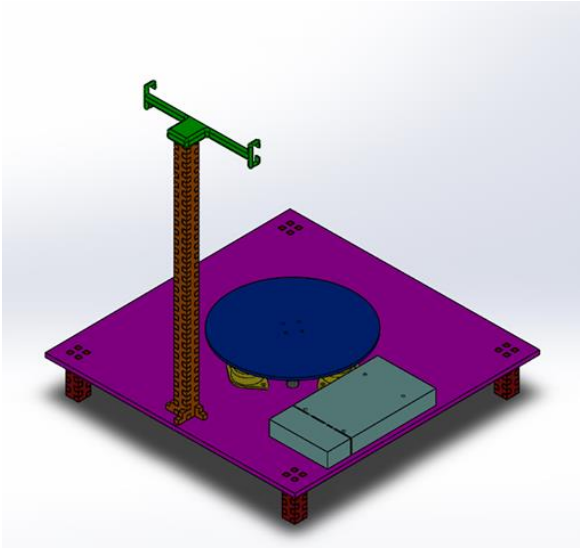


Figure 10. Initial prototype

However, after the initial design process, when we wished to manufacture the parts, we were informed the laser cutter is unavailable because students from another module were using it for the whole week.
After that initial setback, we most likely would have continued with our original design if the laser cutter did not break just after becoming available and set us back even further.

Team decision was made to switch manufacturing process to 3D printing.
A new design was produced in SolidWorks, using top down methodology, with added features that became possible with 3D printing.

After some experimenting and testing (documented in "Testing/Demonstration" chapter), necessary modifications were made, like the axle brace pictured below and we settled on the optimal assembly for the turret.

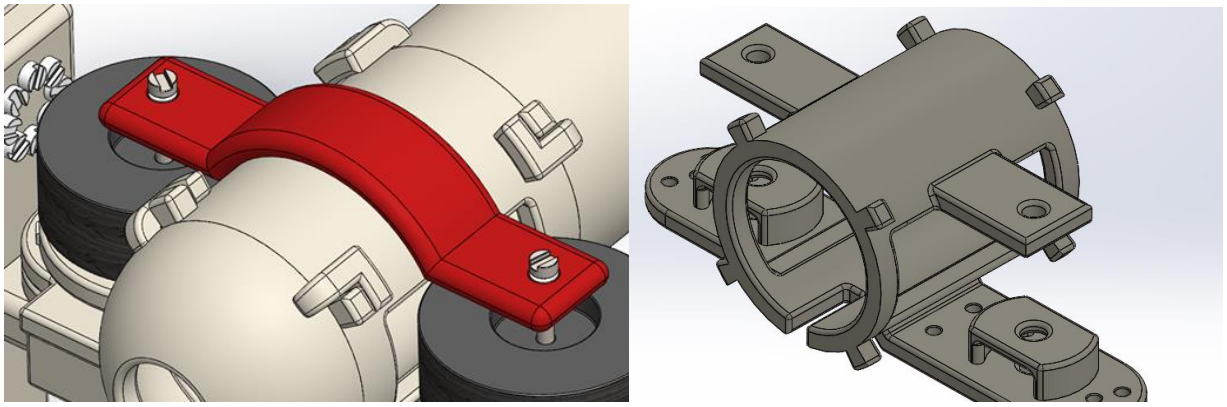
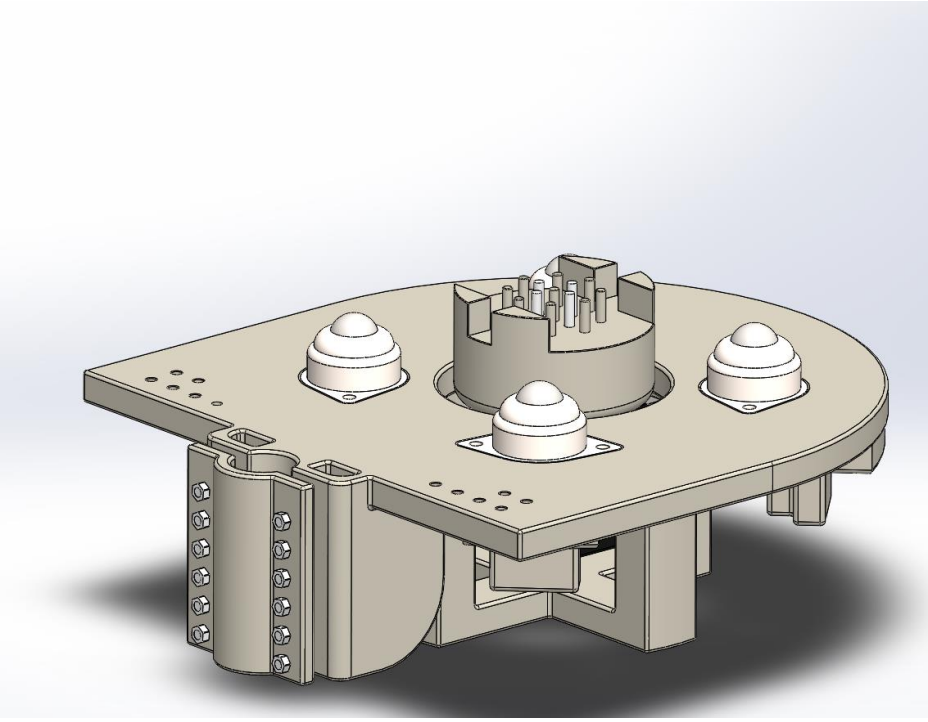
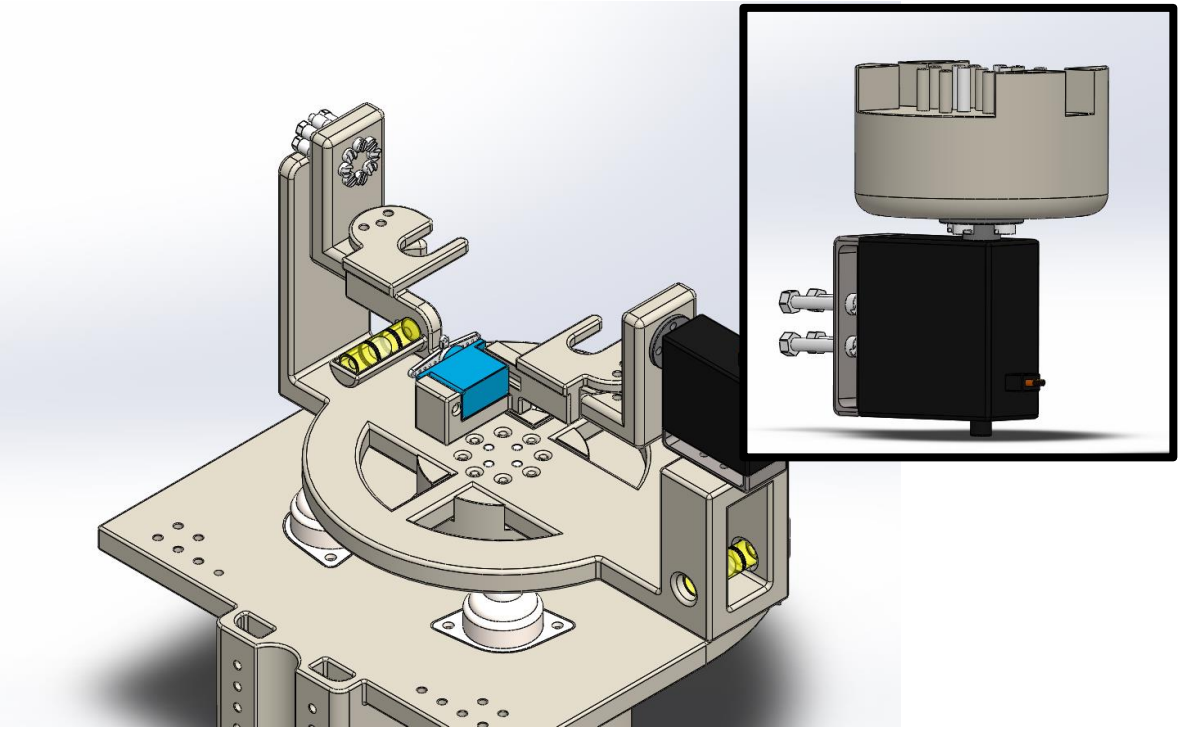


Figure 11. Experimental brace (red) which proved useful and was redesigned into the finished product as the two wings at the top of the launch chamber.

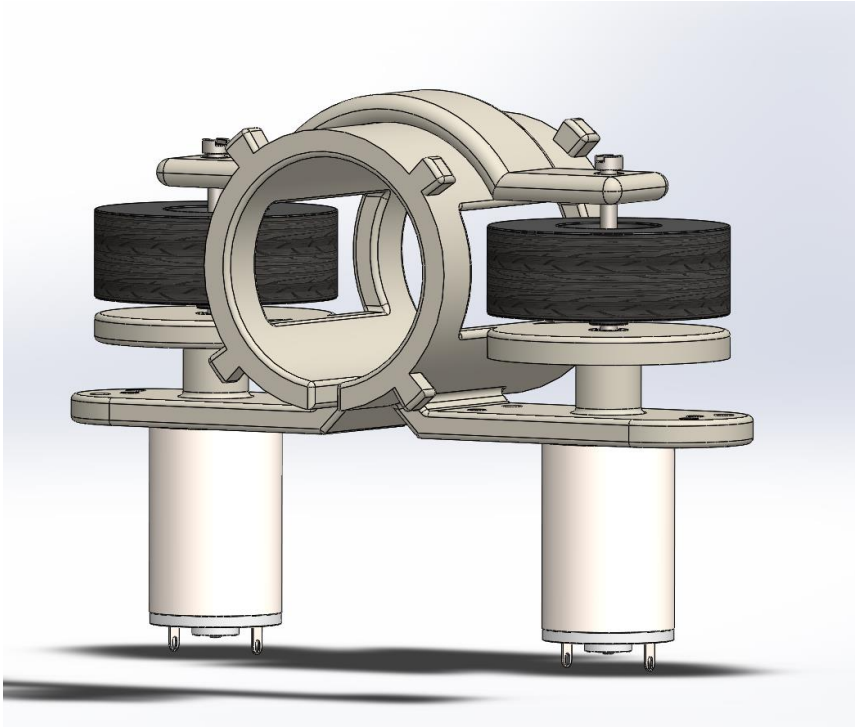
Starting at the bottom, the base has mounting for a metal pole on which the camera sits as well as housing for an azimuth servo to rotate the assembly horizontally and four roller bearings to make the rotation smooth. This mechanism is linked to the rest of the assembly via an adapter (seen protruding in the middle).



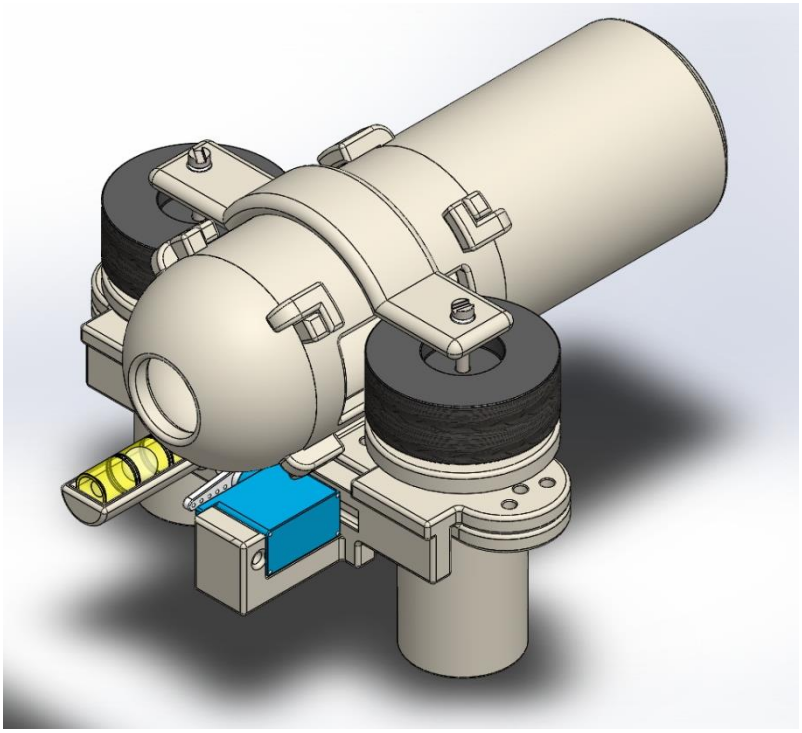
On top of the base sits the elevation mechanism with a small servo that will push the ball into the fly-wheels and couple of spirit level bubbles to allow us to easily see when the barrel is horizontal.



On top of that sits the launch mechanism which consists of launch chamber where the ball is fired from by the fly-wheels that are powered by DC motors just below.

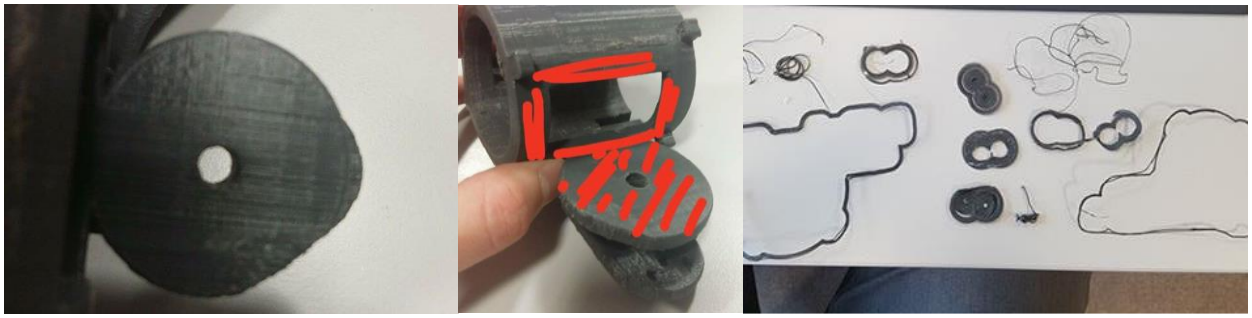


A barrel and a breech can be fitted to the launch chamber via a twist lock mechanism.
The breech (hollow semi spherical object) is designed to hold the ball before it is pushed in between the wheels to be fired and the barrel is to improve the accuracy. The barrel length was calculated so that the total guided path of the ball from the wheels to the exit of the barrel is 2.5 times the diameter of the ball. This along with miniscule clearance (manually corrected with a file) will provide suitable accuracy since there is no point to having a helical rifling to the barrel as the projectile is round and would not benefit from the spin effect.

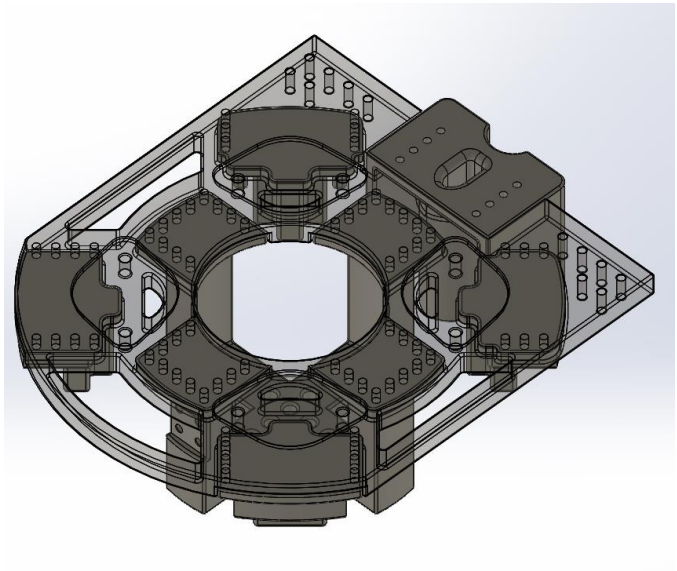


The whole assembly is held together by nylon screws for non-critical sections and steel screws where strength & rigidity is required (wheel axles).

With 3D printing we have run into limitations and had to take into account special requirements when designing. Notable ones are volume/strength ratio, material, size & build time and the most notorious of all, part warping, which ended up causing us constant mission critical problems (Part warping and build failure pictured below).



After consultation, to avoid warping, we have split some of the larger parts such as the base into more smaller parts. To counter the possible misalignment, a high accuracy TAS6 printer was used for manufacture, and a fleet of screws for securing parts to one another.



Finally, by borrowing a printer for home use from robotics department and working with friends, course colleagues and staff from the university we were able to manufacture all the necessary parts with desired tolerances and without warping.

We would like to extend special thanks to Séverin Lemaignan, Martin Stolen, Jake Shaw-Sutton, Bob Williams, Tom Queen, Ben Green & Wes Andrews, for their advice & manufacturing help without whom we would not be able to produce this high-quality work in such a short time when the 3D printers were incredibly busy with other peoples work.

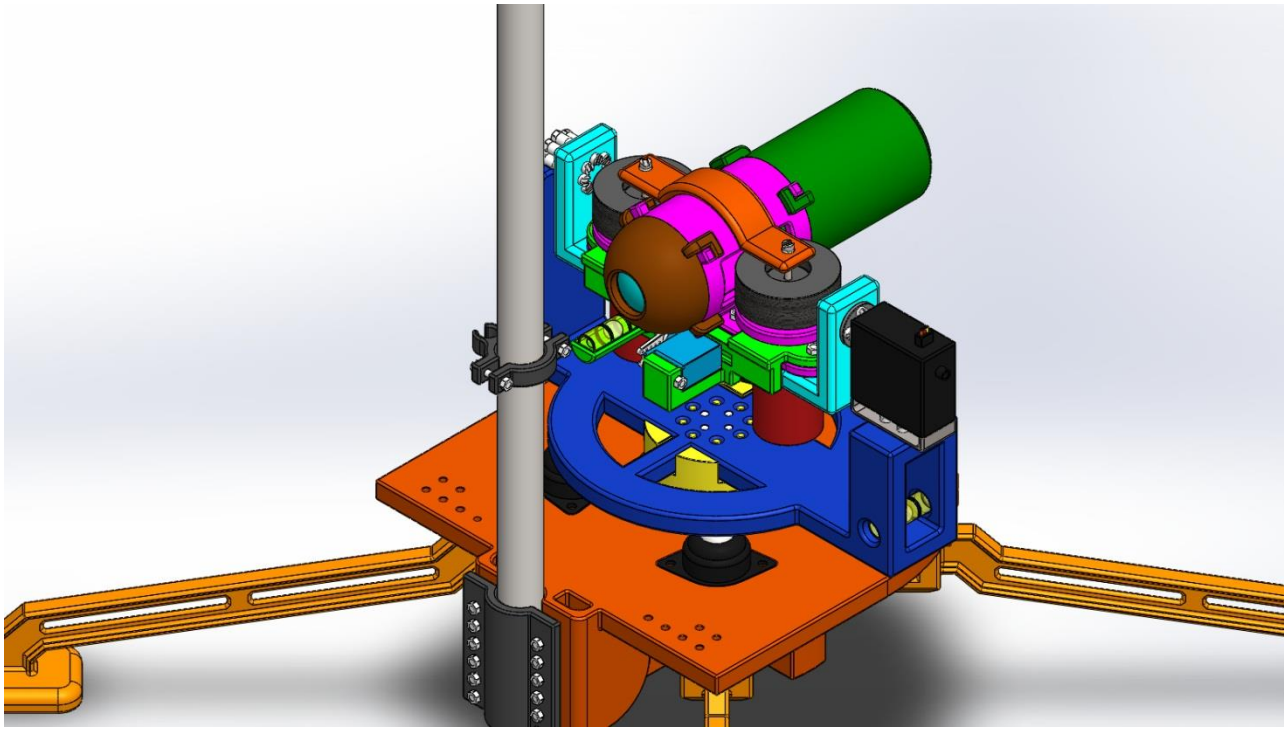


Figure 12. The assembly, once all the bits are attached (colour coded by part).

2.7 Procurement & Manufacture

We encountered a number of difficulties when making the turret. These largely boiled down to long delivery times for parts, and lack of availability for equipment.

Procurement

Most of the parts bought were intentionally better than needed. Given the time constraint for the project, and the low cost of parts, it was deemed prudent to spend a little more on parts that are guaranteed to perform.

HJ S3315D servos [11]

The HJ servos have a range of 180 degrees, can produce 14kg.cm of torque at 6volts, and feature metal gears and mountings. At a price of under £10 each, they are well suited for the turret, as they are more than capable of moving the mass to a specific position, and holding that position.

Rapid high torque DC motors [12]

Nominal voltage 1.5 to 6v range (3v recommended) outputting 1.58W, max stall current 5.2A. Crucially, the motors can output 19g.cm of torque at 3000rpm, with a 6v input. Again, more than enough for the task.

L293D Half-H Motor Driver

Output current 600mA per channel. This is enough when the DC motors aren't running at full speed. If they are, then a bench supply is needed as the current draw is 900mA per motor.
Wide supply 4V5-36V.
Features enable pins for PWM control. Drivers are enabled in pairs.

15mm ball conveyer roller bearings [13]

Low quality ball rollers, used to take the weight off of the horizontal rotation servo motor. They don't feature a low rolling resistance, but that isn't needed for the task.

Screws:
35mm, M3, steel, Philips round head screws.
30mm, M3, nylon6, flat cheese head screws.

3. Testing/Demonstration

3.1 Testing

We quickly found when testing the early code that the HSV settings for the ball and target needed re-calibrating for every environment, and in environments where natural light was a factor, re-calibrating every few hours. Additionally, other environment factors had to be taken into consideration, such as the colour of walls, tables, or computer screens. This is a natural downside machine vision in uncontrolled environments.

When the ball shooting mechanism was assembled with the 3D printed parts, we found the tolerances for the barrel diameter, the gaps for the wheels were too tight. Because of this, the ball did not smoothly fit into the barrel, and when the wheels made contact with the 3D printed part, the friction caused the wheel to unscrew itself from the motor-screw adapter. These parts were then filed and cut to size, and some subsequent changes were made to the design of the part.

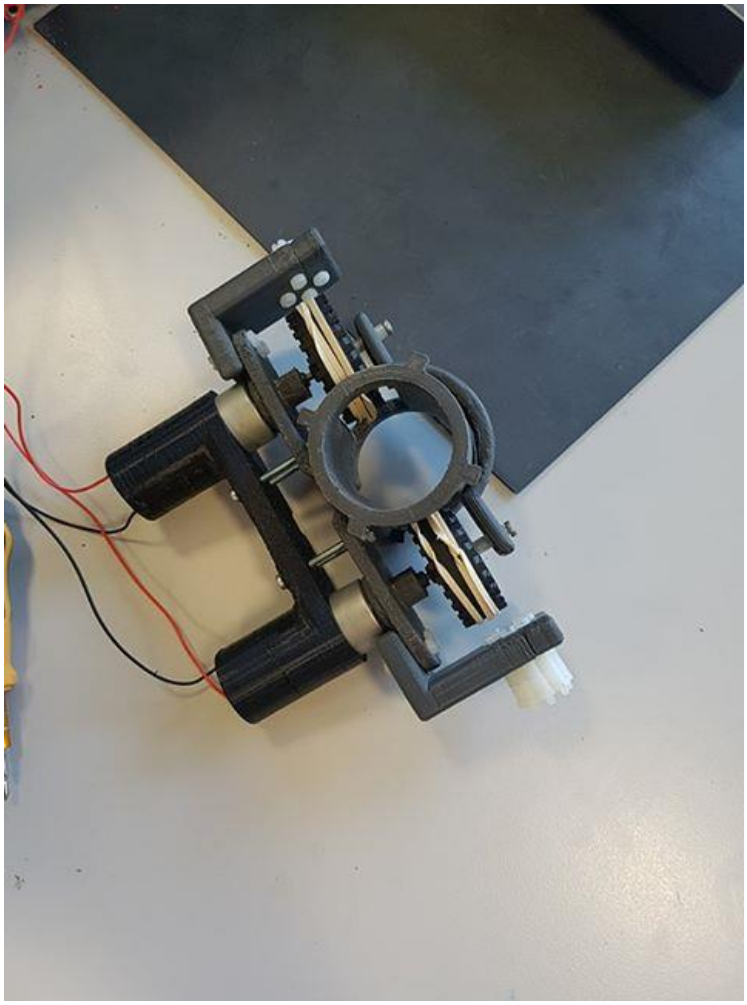


Figure 13. The first prototype, after being modified to reduce contact of the wheels to the barrel [Link to video].

3.2 Project Demonstration

As we did not have the full turret parts printed and assembled, we had to adjust our demonstration.

- The CAD files, showing what the final turret will look like, once parts had been manufactured and assembled.
- The ball tracking code, showing the X & Y location on the screen
- The target tracking code, including X,Y & Z (we explained how we estimated the distance of the target)
- The trajectory calculations and graphical plot
- The servos moving, controlled by a Nucleo board, itself receiving position instructions from a computer.
- The hit/miss code, showing that we could determine whether the ball had hit or miss the target.
- The ball firing mechanism shooting a ball, with the dc motors running off a power supply.

3.3 Further development and Testing

Following from the demonstration, we as a team felt passionate enough for the project to see it through to completion. This final section details the continued development of the turret.

All the parts for the turret were printed during November, including the redesigned barrel.

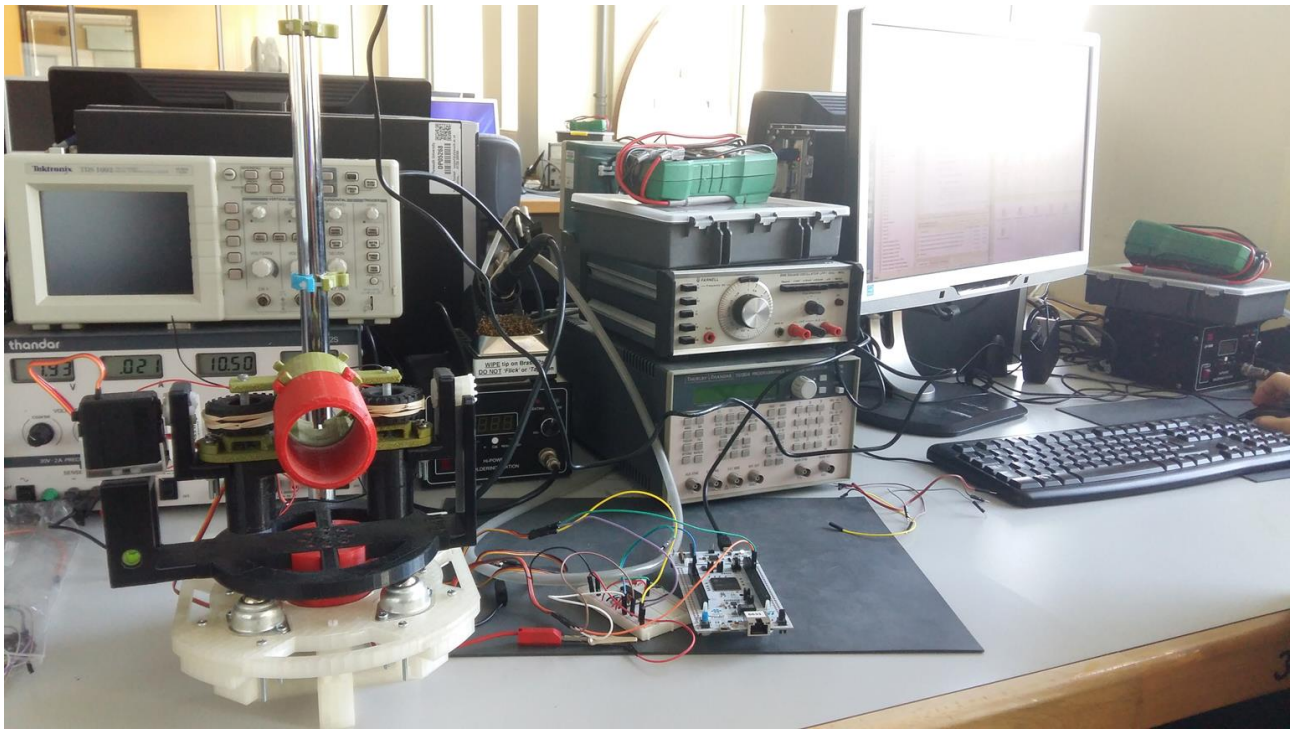


Figure 14. Photo of the assembled turret

With the turret assembled, we then tested the turrets ability to track a target, its shooting accuracy, and measured the initial velocity of the ball.

Figure 15. Testing the accuracy of the shooting mechanism [Link to Video].

As seen in in figure 15 by the grouping of the marks, the turret hit the target 5/6 times, with two very tight groupings of two. We concluded that the variation was likely a result of feeding the balls in manually, and that a servo controlled ball feeder would largely eliminate the errors.

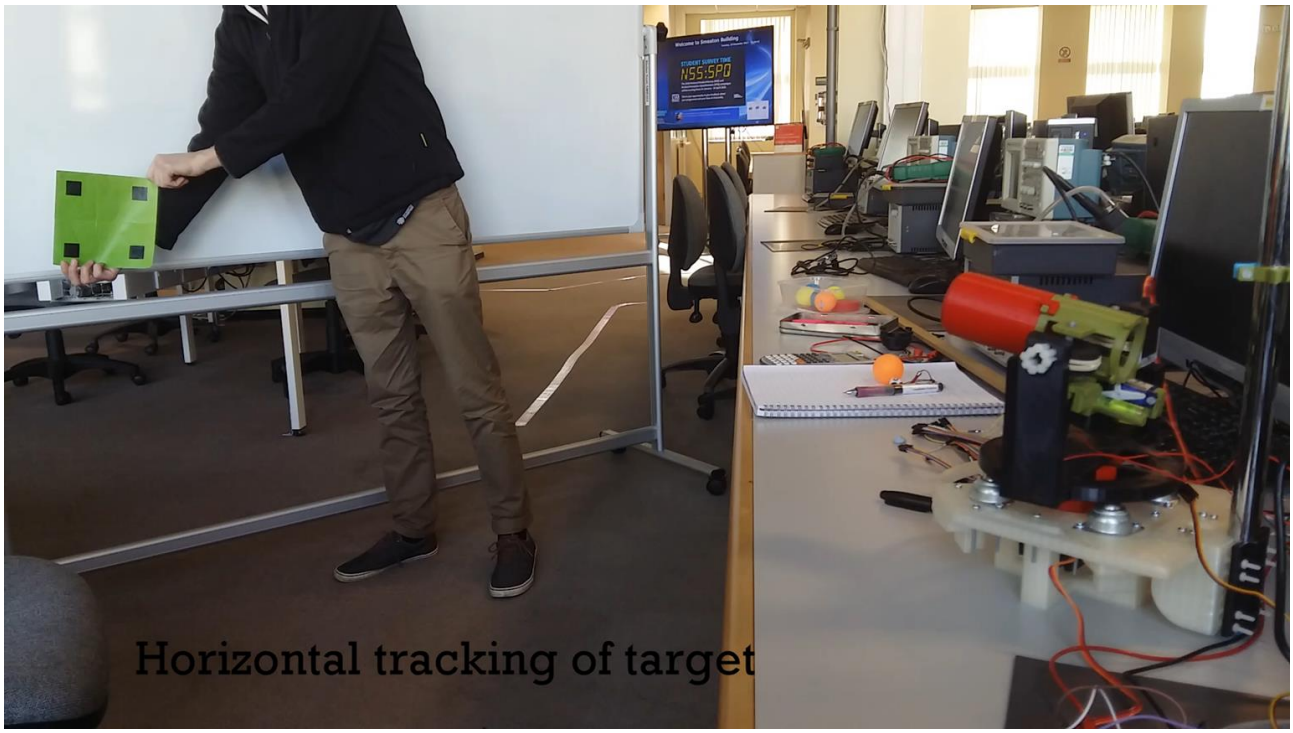


Figure 16. The turret tracking the target horizontally [Link to Video].

The horizontal tracking worked (Figure 16), but we found that occasionally the turret would move to a new position and then stutter. We believe the cause of this is sending the servo position (a value between 0 & 1) as a large floating point, and that the servo was not always able to resolve the input.

The speed was estimated by taking rapid fire photos of the camera frame, with a program Elliott wrote, takesnapshot.py, and calculating the distance travelled over the frames.

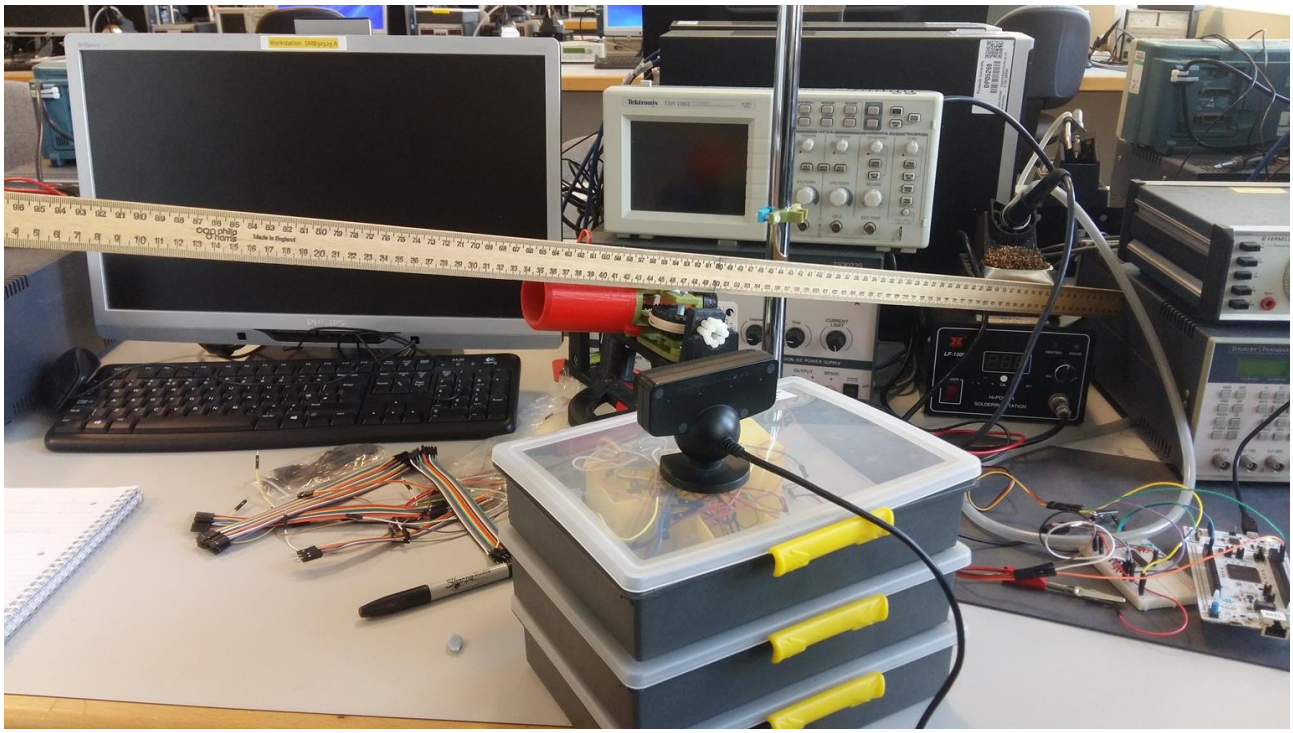


Figure 17. The setup for measuring the initial speed of the ball.

Figure 18. A composition of the ball leaving the turret.

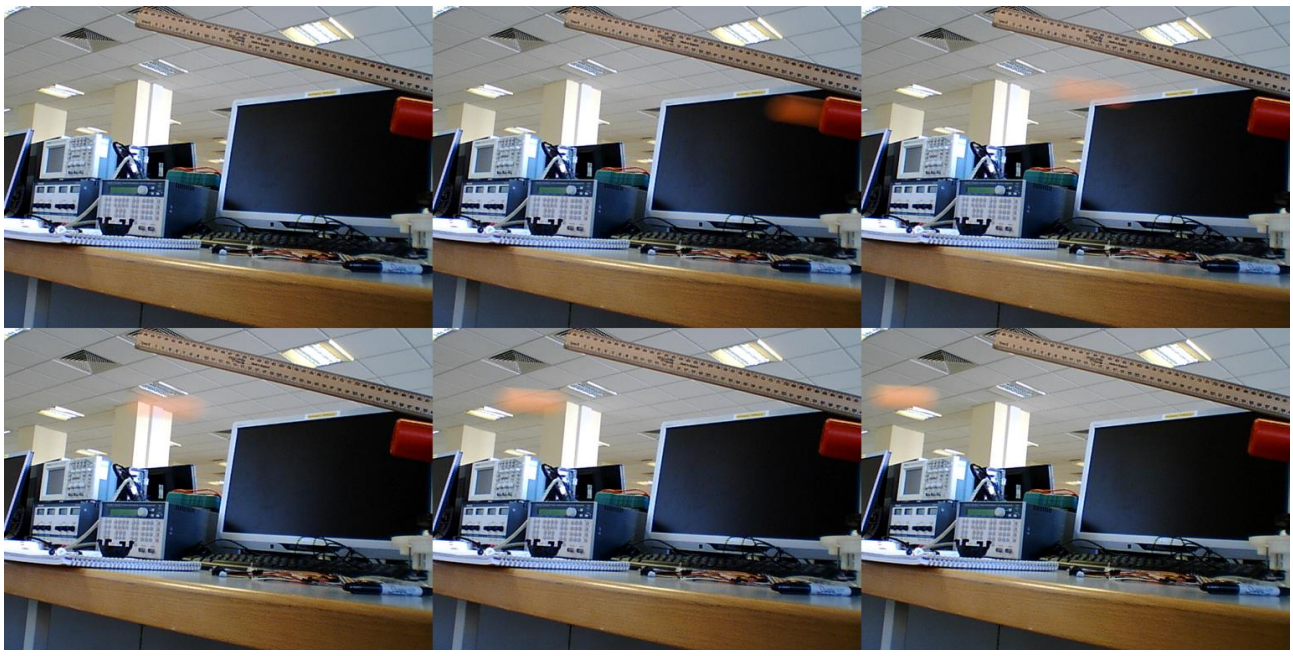


Figure 19. Multiple frames of the ball leaving the turret.

The speed of the ball was measured three times. In the first test, the ball travelled 29cm in 2 frames, filmed at 60fps (1 frame/60 fps = 0.0166seconds). Thus, 0.29m / (2 * 0.0166) = 8.73m/s as an initial velocity.

A second and third test saw speeds of 6.9m/s and 7.4m/s. The resulting average initial velocity was 7.676m/s. As with the testing of the shooting accuracy, we suspect the variance will reduce with a servo controlled ball feeder.

4. Conclusions

4.1 The Turret

At the time of writing, the turret still needs minor additional development to meet the goals we initially set. We are confident in its mechanical performance and that the code will yield the results we are after.

On reflection, we recognise that the first prototypes were designed with high tolerances, and that a prototype only needs to achieve the minimum goals for a proof of concept. However, with further development this did lead to a very well designed and manufactured robot. Additionally, the team dealt with numerous software, procurement, and manufacturing setbacks. In particular, while the Robot Operating System was not best suited to our application given the time frame and steep learning curve, we understand its value, and can see the potential of its application to our project when dealing with more complex tasks.

4.2 The sensor

The Playstation EYE camera has proven itself to be a valuable tool for machine vision applications. Capable of reliably delivering a framerate above anything else on the market for the price. Our project never attempted to utilise the 4 microphone, which itself may prove to be a useful robotics development tool.

Sadly the sensor is no longer commercially manufactured, and this limits its wider use as a commercial part. However, it is still an ideal camera for machine vision purposes, and will remain useful in rapid prototyping and academic research applications.

5. Bibliography

[1] The Playstation EYE manual (https://www.playstation.com/en-ie/content/dam/support/manuals/scee/web-manuals/peripherals/ps3/ps3-eye/SCEH-00448_PS%20Eye%20Web_GB.pdf/)
[2] Omnivison OV7221 Datasheet (http://www.zhopper.narod.ru/mobile/ov7720_ov7221_full.pdf)
[3] Windows driver for the EYE (https://codelaboratories.com/downloads)
[4] Interview with Sony UK VP (https://web.archive.org/web/20120309180342/http://www.officialplaystationmagazine.co.uk:80/2012/03/08/sony-boss-on-move-we-could-have-done-a-better-job/)
[5] CEX link to the Playstation EYE (https://uk.webuy.com/product.php?sku=spseyee001)
[6] Raspberry Pi camera V2 (https://www.raspberrypi.org/products/camera-module-v2/)
[7] Matrix vision mvBlueFox (https://www.matrix-vision.com/USB2.0-industrial-camera-mvbluefox.html)
[8] ROS.org (http://www.ros.org/)
[9] Baxter (https://en.wikipedia.org/wiki/Baxter_(robot))
[10] Robotnaut (https://en.wikipedia.org/wiki/Robonaut)
[11] HJ S3315D servos (https://www.amazon.co.uk/GoolRC-Performance-Brushed-Rotating-Straight/dp/B00YWLS26M)
[12] Rapid high torque DC motors (https://www.rapidonline.com/rapid-high-torque-electric-motor-3-6v-07-0003)
[13] 15mm ball conveyer roller bearings (https://www.amazon.co.uk/gp/product/B01D4XB9D6/ref=oh_aui_detailpage_o01_s00?ie=UTF8&psc=1)

6. Appendix

[1] The turret code “ball_and_square_v3”

```
1.  import cv2
2.  import numpy as np
3.  import serial
4.  import time
5.  import sys
6.  from serial import SerialException
7.  import struct
```

```
import math
import matplotlib.pylab as plot
import argparse
from collections import deque

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video",
    help="path to the (optional) video file")
ap.add_argument("-b", "--buffer", type=int, default=64,
    help="max buffer size")
args = vars(ap.parse_args())


cap = cv2.VideoCapture(0)


# SET FRAME SIZE AND FRAME RATE
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FPS, 60)


# OBJECT COLOR RANGES
color_ranges = [
    ((33,43,115),(98,255,175), 'square'),
    ((96,148,136),(130,255,255), 'ball')]


# IMAGE DISTANCE CALIBRATION DATA
KNOWN_DISTANCE = 30
KNOWN_WIDTH = 20
IMAGE_PATH = "/home/elliott/turret/30cm_image_2.png"    # Calibration im-
age may be slightly off

BALL_TO_FLOOR_DISTANCE = 22.5 #CM
#CAMERA_TO_FLOOR_DISTANCE = 30 #CM
CAMERA_TO_BALL_DISTANCE = 40 #CM

# INITIALISE FRAME COUNTER
frameCounter = 0

# INITIALISE GRAPH
graph1 = plot.figure()
v = 5           # initial velocity of the ball. Declared as we don't know it

# INITIALISE VALUES FOR HIT OR MISS TIMERS
startTime = 0
timeToLaunch = 0.32     #Value in seconds for ball to be launched out of turret

pts = deque(maxlen=args["buffer"])


# FUNCTION FOR INTIAL CALIBRATION TO FIND THE TARGET DISTANCE
def find_marker(image):
    # convert the image to grayscale, blur it, and detect edges
    #gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    #gray = cv2.GaussianBlur(gray, (1, 1), 0)
    #edged = cv2.Canny(gray, 0, 100)

    # find the contours in the edged image and keep the largest one;
    # we'll assume that this is our piece of paper in the image
    #contours = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
       #cv2.CHAIN_APPROX_SIMPLE)[-2]
    #c = max(contours, key = cv2.contourArea)

    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    mask = cv2.inRange(hsv, (38,28,65),(84,162,255))
```

```
76.
77.         # Additional thresholding, erode & dilate, removes noise
78.     mask = cv2.erode(mask, None, iterations=0)
79.     mask = cv2.dilate(mask, None, iterations=0)
80.
81.         # Bitwise-AND mask and original image
82.     res = cv2.bitwise_and(image,image, mask= mask)
83.
84.
85.         # Find contours (A ball)
86.     contours = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
87.             cv2.CHAIN_APPROX_SIMPLE)[-2]
88.
89.     c = max(contours, key=cv2.contourArea)
90.     rect = cv2.minAreaRect(c)
91.     box = cv2.boxPoints(rect)
92.     box = np.int0(box)
93.     cv2.drawContours(image, [box], 0, (0, 255, 255), 2)
94.
95.
96.     # compute the bounding box of the of the paper region and return it
97.     return cv2.minAreaRect(c)
98.
99.
100.        # FUNCTION TO CALCULATE DISTANCE OF TARGET FROM CAMERA
101.        def distance_to_camera(knownWidth, focalLength, perWidth):
102.
103.            if perWidth != 0:
104.                # compute and return the distance from the maker to the camera
105.                return (knownWidth * focalLength) / perWidth
106.            else:
107.                return 0
108.
109.
110.        # FUNCTION TO CALCULATE DISTANCE OF TARGET FROM CENTER OF FRAMCE
111.        def distance_from_center(center_x_pos, knownWidth, perWidth):
112.
113.            if perWidth != 0:
114.                if center_x_pos > 320:
115.                    distance_from_center = center_x_pos - 320
116.                else:
117.                    distance_from_center = -(320 - center_x_pos)
118.
119.                if perWidth != 0:
120.                    cm_per_pixal = knownWidth / perWidth
121.                else:
122.                    cm_per_pixal = 0
123.
124.                return distance_from_center * cm_per_pixal
125.            else:
126.                return 0
127.
128.
129.        # FUNCTION TO CALCULATE DISTANCE OF TARGET TO TURRET
130.        def distance_to_turret_calc(dis_to_cam_cm):
131.
132.            #a**2 + b**2 = c**2
133.
134.            dis_to_turr_base_sq = (dis_to_cam_cm ** 2) - ((BALL_TO_FLOOR_DISTANCE + CAM-
    ERA_TO_BALL_DISTANCE) ** 2)
135.
136.            if (dis_to_turr_base_sq > 0):
137.                dis_to_turr_sq = dis_to_turr_base_sq + (BALL_TO_FLOOR_DISTANCE ** 2)
138.                return math.sqrt(dis_to_turr_sq)
139.            else:
140.                return 0
141.
142.        # FUNCTION TO CALCULATE DISTANCE OF TARGET TO TURRET BASE
143.        def distance_to_turret_base_calc(dis_to_turret):
```

```
144.
145.             dis_to_turr_base_sq = (dis_to_turret ** 2) - (BALL_TO_FLOOR_DISTANCE ** 2)
146.
147.             if dis_to_turr_base_sq > 0:
148.
149.                 return math.sqrt(dis_to_turr_base_sq)
150.
151.             else:
152.                 return 0
153.
154.
155.         # FUNCTION TO CALCULATE THE ySERVO ANGLE
156.         def calculate_yservo_angle(dis_to_turret_base_m):
157.
158.             #Calulated using equation on Wikipedia
159.             BALL_TO_FLOOR_DISTANCE_M = BALL_TO_FLOOR_DISTANCE / 100
160.
161.             tmp1 = (9.81 * (dis_to_turret_base_m ** 2)) + (2 * BALL_TO_FLOOR_DIS-
    TANCE_M * (v ** 2))
162.
163.             if (9.81 * tmp1) < (v ** 4):
164.                 tmp2 = math.sqrt((v ** 4) - (9.81 * tmp1))
165.                 tmp3 = ((v ** 2) + tmp2) / (9.81 * dis_to_turret_base_m)
166.                 theta = math.atan(tmp3)
167.
168.                 return theta
169.             else:
170.                 return 0
171.
172.
173.         # FUNCTION TO CALCULATE THE TIME OF FLIGHT
174.         def calculate_time_of_flight(theta, BALL_TO_FLOOR_DISTANCE):
175.
176.             #Calculated using equation on Wikipedia
177.             return (2 * v * math.sin(theta)) / 9.81
178.
179.
180.
181.         # FUNCTION TO DRAW TRAJECTORY GRAPH
182.         def draw_graph(theta):
183.
184.             #theta = math.pi/2 - theta
185.
186.             t = np.linspace(0, 2, num=500) # Set time as 'continous' parameter.
187.
188.             #for i in theta: # Calculate trajectory for every angle
189.             x1 = []
190.             y1 = []
191.             for k in t:
192.                 x = ((v*k)*math.cos(theta))
193.                 y = ((v*k)*math.sin(theta))-((0.5*9.81)*(k*k))
194.                 x1.append(x)
195.                 y1.append(y)
196.             p = [i for i, j in enumerate(y1) if j < 0]
197.
198.             for i in sorted(p, reverse = True):
199.                 del x1[i]
200.                 del y1[i]
201.
202.             plot.axis([0.0,2.5, 0.0,2.0])
203.             ax = plot.gca()
204.             ax.set_autoscale_on(False)
205.
206.             theta_deg = math.degrees(theta)
207.             plot.plot(x1, y1, label='yServoAngle = %f' % theta_deg) # Plot for every an-
    gle
208.             plot.legend(loc='upper right')
209.             plot.ion()
210.             plot.pause(0.000000001)
```

```
211.            plot.draw() # And show on one graphic
212.            graph1.clear()
213.
214.
215.        # FUNCTION TO DRAW CIRCLE AROUND BALL
216.        def draw_circle(contours, center):
217.            # If contours are found
218.            if len(contours) > 0:
219.                # find the largest contour in the mask, then use
220.                # it to compute the minimum enclosing circle and
221.                # centroid
222.                c = max(contours, key=cv2.contourArea)
223.                ((x, y), radius) = cv2.minEnclosingCircle(c)
224.                M = cv2.moments(c)
225.                # This line calculates centroid
226.                if M["m00"] != 0:
227.                    draw_circle.cen-
    ter = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
228.
229.                    # update the points queue
230.                    pts.appendleft(draw_circle.center)
231.
232.                    if radius > 1:
233.                        #cv2.circle(img, center, radius, color, thickness=1, line-
    Type=8, shift=0)
234.                        cv2.circle(frame, (int(x), int(y)), int(radius),(0, 255, 255), 2)
235.                        cv2.circle(frame, draw_circle.center, 5, (0, 0, 255), -1)
236.                        #print('circle', draw_circle.center)
237.
238.                    # loop over the set of tracked points
239.                    for i in xrange(1, len(pts)):
240.                        # if either of the tracked points are None, ignore
241.                        # them
242.                        if pts[i - 1] is None or pts[i] is None:
243.                            continue
244.
245.                        # otherwise, compute the thickness of the line and
246.                        # draw the connecting lines
247.                        #thickness = int(np.sqrt(args["buffer"] / float(i + 1)) * 2.5)
248.                        thickness = 5
249.                        cv2.line(frame, pts[i - 1], pts[i], (0, 0, 255), thickness)
250.
251.
252.        # FUNCTION TO DRAW RECTANGLE AROUND TARGET AND CALCULATE ANGLE TO SEND TO xSERVO
253.        def draw_rect(frameCounter):
254.            # If contours are found
255.            if len(contours) > 0:
256.                c = max(contours, key=cv2.contourArea)
257.                rect = cv2.minAreaRect(c)
258.                box = cv2.boxPoints(rect)
259.            box = np.int0(box)
260.                cv2.drawContours(frame, [box], 0, (0, 255, 255), 2)
261.
262.                # box prints out array as the following:
263.            #[ [bottom left  (x,y)]
264.            #  [top left     (x,y)]
265.            #  [top right    (x,y)]
266.            #  [bottom right (x,y)] ]
267.
268.            # box[X] can be assigned to their corresponding positions
269.            draw_rect.bl_pos = box[0]
270.            draw_rect.tl_pos = box[1]
271.            draw_rect.tr_pos = box[2]
272.            draw_rect.br_pos = box[3]
273.
274.            # Each position is an array with X and Y coordinates
275.            # Now calculate the center X coordinate
276.            center_top_x_pos = (draw_rect.tl_pos[0] + draw_rect.tr_pos[0]) / 2
277.            center_bottom_x_pos = (draw_rect.bl_pos[0] + draw_rect.br_pos[0]) / 2
```

```
            center_x_pos = (center_top_x_pos + center_bottom_x_pos) / 2
            #print(center_x_pos)

            # Now calculate the center Y coordinate
            center_left_y_pos = (draw_rect.tl_pos[1] + draw_rect.bl_pos[1]) / 2
            center_right_y_pos = (draw_rect.tr_pos[1] + draw_rect.br_pos[1]) / 2
            center_y_pos = (center_left_y_pos + center_right_y_pos) / 2
            #print('square', center_x_pos, center_y_pos)

            # Draw center point
            cv2.circle(frame, (center_x_pos, center_y_pos), 8, (255, 0, 0), -1)



            # Calculate distance of rectangle to mid-
    dle of the frame. rect[1][0] is the width of the rectangle in pixals
            dis_from_center_cm = distance_from_center(cen-
    ter_x_pos, KNOWN_WIDTH, rect[1][0])

            # Calculate distance of rectangle to cam-
    era. rect[1][0] is the width of the rectangle in pixals
            dis_to_cam_cm = distance_to_camera(KNOWN_WIDTH, focalLength, rect[1][0])

            dis_to_turr_cm = distance_to_turret_calc(dis_to_cam_cm)

            dis_to_turr_base_cm = distance_to_turret_base_calc(dis_to_turr_cm)

            dis_to_turr_base_m = dis_to_turr_base_cm / 100


            cv2.putText(frame, "Distance to cam-
    era: %.2fcm" % dis_to_cam_cm,(5,10),cv2.FONT_HERSHEY_COMPLEX_SMALL,.5,(225,0,0))
            cv2.putText(frame, "Distance from center: %.2fcm" % dis_from_cen-
    ter_cm,(5,25),cv2.FONT_HERSHEY_COMPLEX_SMALL,.5,(225,0,0))
            cv2.putText(frame, "Distance to tur-
    ret base: %.2fcm" % dis_to_turr_base_cm,(5,40),cv2.FONT_HERSHEY_COM-
    PLEX_SMALL,.5,(225,0,0))

            # Calculate the angle needed for the xServo
            if ((dis_from_center_cm / dis_to_turr_cm) < 1) and ((dis_from_cen-
    ter_cm / dis_to_turr_cm) > -1):
                if dis_to_turr_cm != 0:
                    xServoAngle = math.asin(dis_from_center_cm / dis_to_turr_cm)
                    xServoAngle = math.degrees(xServoAngle)
            else:
                xServoAngle = 0

            # Calculate the angle needed for the yServo
            if dis_to_turr_base_cm > 0:

                theta = calculate_yservo_an-
    gle(dis_to_turr_base_m)  # Need to change to take into account the height of turret be-
    fore this
                theta = math.degrees(theta)

                if theta != 0:
                    yServoAngle = theta

                    draw_rect.time_of_flight = calcu-
    late_time_of_flight(theta, BALL_TO_FLOOR_DISTANCE)

                    cv2.putText(frame, "Time of flight: %.2fS" % draw_rect.time_of_flight,(
    5,55),cv2.FONT_HERSHEY_COMPLEX_SMALL,.5,(225,0,0))

                    if frameCounter == 120:
                        theta = math.radians(theta)
                        draw_graph(theta)
                else:
                    print("ERROR - TARGET TOO FAR")
```

```
335.                     yServoAngle = 0
336.
337.             else:
338.                 print("ERROR - TARGET TOO CLOSE")
339.                 yServoAngle = 0
340.
341.
342.
343.             # Write the angles to the Servos every 60 frames
344.             if frameCounter == 60:
345.                 xServoAngle_as_string = str(xServoAngle)
346.                 ser.write(b'xServo')
347.                 print('xServo')
348.                 time.sleep(0.03)
349.                 ser.write(xServoAngle_as_string)
350.                 print(xServoAngle_as_string)
351.                 time.sleep(0.03)
352.
353.             if frameCounter == 120:
354.                 center_y_pos_as_string = str(yServoAngle)
355.                 ser.write(b'yServo')
356.                 print('yServo')
357.                 time.sleep(0.03)
358.                 ser.write(center_y_pos_as_string)
359.                 print(center_y_pos_as_string)
360.                 time.sleep(0.03)
361.
362.
363.         # FUNCTION TO DETECT IF BALL HAS HIT TARGET
364.         def hit_box():
365.             # if ball_x is in range of square x1, x2?
366.             print(draw_circle.center)
367.             if draw_circle.center != 0:
368.                 if draw_rect.tl_pos[0] < draw_circle.cen-
    ter[0] < draw_rect.tr_pos[0] and draw_rect.tl_pos[1] < draw_circle.cen-
    ter[1] < draw_rect.bl_pos[1]:
369.                     # and if ball_Y is in range of square y
370.                     print('HIT!')
371.                 # Doesn't account for square rotation...
372.                 else:
373.                     print('MISS!')
374.
375.
376.
377.         # CALIBRATE CAMERA
378.         image = cv2.imread(IMAGE_PATH)
379.         marker = find_marker(image)
380.         focalLength = (marker[1][0] * KNOWN_DISTANCE) / KNOWN_WIDTH
381.
382.         centimetres = distance_to_camera(KNOWN_WIDTH, focalLength, marker[1][0])
383.
384.         box = cv2.boxPoints(marker)
385.         box = np.int0(box)
386.         cv2.drawContours(image, [box], -1, (0, 255, 0), 2)
387.
388.         cv2.putText(image, "Distance to camera: %.2fcm" % centimetres,(5,25),cv2.FONT_HER-
    SHEY_COMPLEX_SMALL,.5,(225,0,0))
389.
390.         cv2.imshow("image", image)
391.         cv2.waitKey(0)
392.
393.
394.         # OPEN SERIAL PORT TO NUCLEO BOARD
395.         try:
396.             ser = serial.Serial('/dev/ttyACM0')
397.         except SerialException:
398.             print('Port already open or access is denied')
399.             sys.exit()
400.
```

```
401.        ser.write("")
402.
403.        print(ser.name)
404.
405.
406.
407.        while(1):
408.
409.            # Take each frame
410.            _, frame = cap.read()
411.
412.            # Convert BGR to HSV
413.            hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
414.
415.        # FIND THE OBJECTs AND MASK IT
416.
417.            # define range of BALL color in HSV
418.            #lower_bounds = np.array([55,110,65])
419.            #upper_bounds = np.array([179,255,255])
420.
421.            # Increment frame counter
422.            frameCounter = frameCounter + 1
423.
424.            for (lower, upper, colorName) in color_ranges:
425.                # Threshold the HSV image to get only specific colors
426.                mask = cv2.inRange(hsv, lower, upper)
427.
428.                # Additional thresholding, erode & dilate, removes noise
429.                mask = cv2.erode(mask, None, iterations=0)
430.                mask = cv2.dilate(mask, None, iterations=0)
431.
432.                # Bitwise-AND mask and original image
433.                res = cv2.bitwise_and(frame,frame, mask= mask)
434.
435.        # DRAW CIRCLE & BOX
436.
437.                # Find contours (A ball)
438.                contours = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
439.                    cv2.CHAIN_APPROX_SIMPLE)[-2]
440.                center = None
441.
442.                if colorName == 'ball':
443.                    draw_circle(contours, center)
444.                else:
445.                    draw_rect(frameCounter)
446.
447.        # HIT OR MISS
448.            timeNow = time.time()
449.
450.            if (startTime and draw_rect.time_of_flight) != 0:
451.
452.                if (timeNow - startTime) > timeToLaunch + draw_rect.time_of_flight:
453.                    hit_box()
454.                    startTime = 0
455.
456.
457.        # SHOW THE FRAME
458.
459.
460.            #cv2.imshow('frame',frame)
461.             # These masks block out all but the ball, including the circles.
462.            # solution is to declare circle vars out of if statement
463.            cv2.imshow('mask',mask)
464.            #cv2.imshow('res',res)
465.            cv2.imshow('view', frame)
466.
467.            # Reset frame counter
468.            if frameCounter == 120:
469.                frameCounter = 0
```

```
470.
471.           k = cv2.waitKey(4) & 0xFF
472.           if k == 27:
473.               break
474.           if k == 108:
475.               ser.write(b'launch')
476.               print('LAUNCH!')
477.               start-
    Time = time.time()         #Need to time how long it takes to move ball and it start to fi
    re
478.
479.       cv2.destroyAllWindows()
```

[2] “hsv_slide.py” Used to optimise the hue, saturation, and value threshold boundaries for the target and the ball

```
1.  import cv2
2.  import numpy as np
3.  #optional argument
4.  def nothing(x):
5.      pass
6.  cap = cv2.VideoCapture(0)
7.
8.  # SET FRAME SIZE AND FRAME RATE
9.  cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
10. cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
11. cap.set(cv2.CAP_PROP_FPS, 60)
12.
13. cv2.namedWindow('image')
14.
15. #easy assigments
16. hh='Hue High'
17. hl='Hue Low'
18. sh='Saturation High'
19. sl='Saturation Low'
20. vh='Value High'
21. vl='Value Low'
22.
23. cv2.createTrackbar(hl, 'image',0,179,nothing)
24. cv2.createTrackbar(hh, 'image',0,179,nothing)
25. cv2.createTrackbar(sl, 'image',0,255,nothing)
26. cv2.createTrackbar(sh, 'image',0,255,nothing)
27. cv2.createTrackbar(vl, 'image',0,255,nothing)
28. cv2.createTrackbar(vh, 'image',0,255,nothing)
29.
30. while(1):
31.     _,frame=cap.read()
32.     frame=cv2.GaussianBlur(frame,(5,5),0)
33.     #convert to HSV from BGR
34.     hsv=cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
35.
36.
37.     #read trackbar positions for all
38.     hul=cv2.getTrackbarPos(hl, 'image')
39.     huh=cv2.getTrackbarPos(hh, 'image')
40.     sal=cv2.getTrackbarPos(sl, 'image')
41.     sah=cv2.getTrackbarPos(sh, 'image')
42.     val=cv2.getTrackbarPos(vl, 'image')
43.     vah=cv2.getTrackbarPos(vh, 'image')
44.     #make array for final values
45.     HSVLOW=np.array([hul,sal,val])
46.     HSVHIGH=np.array([huh,sah,vah])
47.
48.     #apply the range on a mask
49.     mask = cv2.inRange(hsv,HSVLOW, HSVHIGH)
```

```
    res = cv2.bitwise_and(frame,frame, mask =mask)

    cv2.imshow('image', res)
    cv2.imshow('yay', frame)
    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break


cv2.destroyAllWindows()
```

[3] “takesnapshot.py” used for measuring the speed of the ball

```
import cv2

# Now we can initialize the camera capture object with the cv2.VideoCapture class.
# All it needs is the index to a camera port.
camera = cv2.VideoCapture(0)

# SET FRAME SIZE AND FRAME RATE
camera.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
camera.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
camera.set(cv2.CAP_PROP_FPS, 60)

#Number of frames to throw away while the camera adjusts to light levels
ramp_frames = 30

burst_frames = 120


# Captures a single image from the camera and returns it in PIL format
def get_image():
 # read is the easiest way to get a full image out of a VideoCapture object.
    retval, im = camera.read()
    return im

# Ramp the camera - these frames will be discarded and are only used to allow v4l2
# to adjust light levels, if necessary
for i in xrange(ramp_frames):
    temp = get_image()
print("Taking image...")

for i in xrange(burst_frames):
# Take the actual image we want to keep
    camera_capture = get_image()
# A nice feature of the imwrite method is that it will automatically choose the
# correct format based on the file extension you provide. Convenient!
    cv2.imwrite('burstimage_' + str(i) + '.png', camera_capture)

# You'll want to release the camera, otherwise you won't be able to create a new
# capture object until your script exits
del(camera)
```

[4] The Playstation EYE manual

PlayStation®Eye

GB

SCEH-00448 7010571

Instruction Manual

Congratulations on your purchase of the PlayStation®Eye camera. Before using the camera, carefully read this manual and retain it for future reference. This camera is designed for use with the PlayStation®3 computer entertainment system.

WARNING

To avoid potential electric shock or starting a fire, do not expose the camera to rain, water or moisture.

Precautions

Safety

This product has been designed with the highest concern for safety. However, any electrical device, if used improperly, has the potential for causing fire, electric shock or personal injury. To ensure accident-free operation, be sure to follow these guidelines.

- Observe all warnings, precautions and instructions.
- Do not use the camera if it functions in an abnormal manner.
- If the camera does not function properly, call the appropriate PlayStation® Customer Service number which can be found:
  – within every PlayStation®, PlayStation®2 and PlayStation®3 format software manual; and
  – on our website PlayStation.com

Using and handling the PlayStation®Eye camera

- Do not expose the camera to high temperatures, high humidity or direct sunlight (operate in an environment where temperatures range between 5°C and 35°C).
- Do not allow liquid or small particles to get into the camera.
- Do not put heavy objects on the camera.
- Never disassemble or modify the camera.
- Do not twist the cable or pull it forcibly.
- Do not throw or drop the camera, or physically damage it any way.
- Do not touch the metal parts or insert foreign objects into the PlayStation®Eye camera's USB connector.
- Do not place or use the camera on an unstable surface.
- Make sure you have enough room to play.

Cleaning the PlayStation®Eye camera

Dust may build up on the camera after an extended period of time.

- Before cleaning the camera, disconnect it from the PlayStation®3 system for safety.
- Wipe the camera's surface with a soft, dry cloth.
- Use air pressure to clean off any dust that has collected on the lens.

Notes

- This product contains small parts, which, if removed, may present a choking hazard to children.
- The camera should be cleaned by an adult, or cleaned under close adult supervision.
- Do not use a damp cloth to clean the camera. If water gets inside, it may cause the camera to malfunction.
- Do not use benzene, paint thinner or other chemicals, as these may damage the camera.
- When using a commercially available cleaning cloth, follow the instructions supplied with the cloth.

Connecting the PlayStation®Eye camera

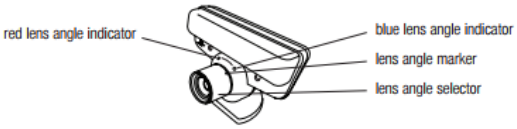
Securely insert the camera's USB connector into one of the USB connectors on the front of the PlayStation®3 system. The blue LED power indicator on the front of the camera should light up to indicate that it is now ready to use.

Disconnecting the PlayStation®Eye camera

To disconnect the camera, gently pull the camera's USB connector out of the PlayStation®3 system. Do not pull on the cable itself as this may damage it. Removal of the camera during play will result in the termination of the live feed.

Using the PlayStation®Eye camera

It is recommended to stand between 1.5 metres and 2.0 metres away from the camera.



Ensure the lens angle selector is rotated to the appropriate setting. Turn the lens angle selector to the right so that the lens angle marker is in line with the blue lens angle indicator for "wide angle view". Alternatively, turn the lens angle selector to the left so that the lens angle marker is in line with the red lens angle indicator for "standard view".

If your camera is positioned on the floor, simply tilt the camera for extra elevation or swivel the camera base around to reposition it. Players of different heights should tilt the angle of the camera for extra convenience. DO NOT move the whole camera, just tilt it gently.

Make sure that the background behind you is as motionless as possible, as background motion may hinder gameplay. Ensure there is enough room to play – watch out for shelves, doors, walls, pets and above all, other people.

For further information about the PlayStation®Eye camera's voice/video chat feature, please refer to the PlayStation®3 System Software User's Guide at PlayStation.com

Troubleshooting

The PlayStation®Eye camera does not recognise your movements.

– You may be experiencing problems with the light in your room.
– Check that there are no flickering lights and that you are evenly lit from the front.
– Avoid using low-energy bulbs and fluorescent tubes if possible and switch on any extra available lights.

The on-screen buttons activate without your interaction.

– There may be background motion behind you.
– Make sure the background behind you is as motionless as possible.
– If there is a window in your background, consider drawing the curtains and using artificial lights instead.

Your on-screen image appears too bright or too dark.

– The lighting in the room is not suitable.
– If your on-screen image looks too dark, switch on all available lights in the room and point them at you.
– If your on-screen image looks too bright, draw the curtains and use artificial light instead.

You cannot see yourself on-screen.

– The camera is not plugged in.
– The camera is not positioned correctly.

Interface	Connector	Power consumption	Dimension (approx.)
USB 2.0	USB	DC5V, Max. 500mA	84 x 67 x 57mm

Weight (approx.)	Cable length (approx.)	Operation temperature	Video capture
173g	2 metres	5°C to 35°C	640 x 480 pixel

Video format	Lenshead
Uncompressed or JPEG	2.1 F-stop, <1% distortion, fixed focus (25cm to ∞ at 75º FOV)

Field of view	Frame rate
56º to 75º FOV zoom lens	640 x 480 at 60 frames/second 320 x 240 at 120 frames/second

Design and specifications are subject to change without notice.