Audio-Visual Object Event Tracking in Time and Space

Robby Rafky

A dissertation submitted to the Department of Electronic Engineering in partial fulfilment of the Degree of Bachelor of Engineering in Computer and Internet Engineering.

Department of Electronic Engineering
Faculty of Engineering and Physical Sciences
University of Surrey
Guildford, Surrey, GU2 7XH, UK

Supervised by Dr P. Jackson

Word Count

38695 Characters

7346 Words

50 Pages

Table of Contents

Table of Figures

Table of Equations

Table of Tables

Abstract

The manipulation of audio has allowed us to create the illusion of audio sources being in distant location as opposed to their true source. Basic visual object tracking methods allow for relatively accurate tracking of small distinct objects in real time. Feeding this location data into real time spatial audio software can give the illusion of these objects producing audio.

Several tracking methods perform differently under different conditions, some much better than others.
For this project, a single camera was used for the tracking of a single object with the aid of binary masking, colour thresholding in the HSV colour space and local frame searching. With the use of the object's real-world dimensions, the software can determine the distance and direction from the camera to the object; mapping this data to an audio source grants the illusion of the tracked object emitting the audio. The main limitations of the software is in the types of objects that can be tracked, larger same-coloured objects getting into the local frame and that only single channel audio can be used.

1. Introduction

1.1 Background and Objectives:

With current day advancements in visual tracking and spatial audio we have achieved high accuracy for real time visual tracking of objects and for simulating audio sources at various locations. In this report, the aim is to setup a system for mapping a tracked object's location to a spatial audio system, imitating that location as the audio source with minimum hardware requirements. For such a system to be made, research and experiments will be done to calculate depth in a 2-Dimensional image from a single camera and the effect of location on audio.
The main objectives for the system are:

- Easy to setup with any camera regardless of its properties and not having any additional hardware requirements.
- Accurate enough to simulate audio in real time without lagging behind real time tracking updates.
- Interpret 3-Dimensional depth from the data of a 2-Dimensional video stream

The first objective to be met will allow for easier testing as it will disconnect the hardware from being anything more than an input for the system, also meaning that the camera should be easily interchangeable with any other cameras to test if there are any desired properties to maximise accuracy.

The second objective to be met is to find a balance between accuracy and performance, as the system will be processing and capturing in real time, higher fidelity image capturing or more complex calculations may cause the system to not be able to keep up with the inflow of real time data.

The third objective to be met is to interpret depth in a 2-Dimensional image given a few properties, this will allow the system to run off a single camera. There are multiple methods to do this, each with varying complexity and differing accuracy.

2. Experimentation and Literature Review

In this chapter a review of the various topics relevant to the project and to building the system were researched and experimented. The following researched topics include brief insight into various tracking algorithms such as; KCF tracker (Kernelized Correlation Filters), TLD tracker (Tracking, Learning and Detection), MedianFlow tracker, MOSSE tracker (Minimum Output Sum of Squared Error), MIL tracker (Multiple Instance Learning), GOTURN tracker (Generic Object Tracking Using Regression Networks) and CSRT (Channel and Spatial Reliability Tracker). Also including research topics that delve into spatial audio properties along with research and experimentation of 2-Dimensional depth interpretations.

2.1 KCF (Kernelized Correlation Filters)

The KCF tracker takes advantage of overlapping regions in multiple positive samples, the data is computed in the Fourier domain to further increase its learning speeds. This tracker uses more samples and emphasizes the significance of negative samples for further accuracy in training, also cyclic shifting is used to extrapolate many more data samples from significant samples. Kernel exploits are used to handle non-linear regression and the tracker extracts the HoG (Histogram of Gradient) features for even more accuracy improvements. [1]

2.2 TLD (Tracking, Learning and Detection)

The TLD tracker is broken down into 3 parts; the tracker, the learner and the detector. The tracker follows the target object across multiple consecutive frames, the learner uses a P-Expert and N-Expert weighting model to estimate false detections and false alarms, then it updates the detector. The detector searches the frame for potential targets based on an appearance model, the learner is fed the output and the tracker is corrected when necessary. The TLD tracker excels at tracking objects under full occlusion for multiple frames at a time, however this tracker is prone target drifting when tracking an object amongst similar objects. [2]

2.3 MedianFlow

The MedianFlow tracker combines forward and backwards tracking in a bidirectional approach to tracking objects with consistency in its movements. The tracker is relatively simple, estimating forward and backwards trajectories at each frame with corresponding errors, the trajectory with the least forward-backwards error is then used with the next frame data to track the object. Due to the innate error estimation, this tracker is easily able to report on tracking failure. [3]

2.4 MOSSE (Minimum Output Sum of Squared Error)

The MOSSE tracker is based on correlation filters, achieving high efficiency by computing correlation in the time domain. It calculates the minimum output sum of the square error to choose the most likely location for the object to be in the current frame. Using a correlation filter allows this tracker to perform well regardless of any deformations in the image or viewport and unlike some other trackers, the tracked object doesn’t need to be in the centre of the image at the start of tracking. [4]

2.5 MIL (Multiple Instance Learning)

The MIL tracker is an extension of an online boosting algorithm (A feature selection algorithm) using bags (collection of image patches) rather than just the single sample training that the boosting algorithm normally uses. Two types of bags are used, positive bags and negative bags, containing positive samples and negative samples respectively. The tracker takes small patches out of the frame centred around the object being tracked as the positive bag to be used for future frames, this allows the MIL tracker to not lose any significant information over the duration of tracking, and avoids any real mislabelling, however this method is vulnerable to inaccuracies after full object occlusion. [5]

2.6 GOTURN (Generic Object Tracking Using Regression Networks)

Through the use of a CNN (Convolutional Neural Network) model, the GOTURN tracker is able to be trained with an offline dataset and as the CNN weighting need not be updated during runtime, the tracker is able to run very efficiently with the appropriate dataset. The GOTURN tracker has a bias towards objects within its offline dataset, however its performance is severely dictated by the quality and relevance of the offline dataset. [6]

2.7 CSRT (Channel and Spatial Reliability Tracker)

The CSRT tracker is based on a DCF (Discriminative Correlation Filter) algorithm, improving on it by introducing channel and spatial reliability. A spatial reliability map is overlaid to optimise the filter size, this can exclude unrealistic shapes for the object being tracked therefore reducing drift in tracking also allowing for the algorithm to better track non-rectangular objects. Channel reliability is a weighting on the significance of each channel filter in the DCF, with the final map being the weighted combination of each filter. This tracker uses HoGs (Histogram of Gradients) and a Colorname standard feature set and with it, the algorithm can perform at real-time speeds while retaining high accuracy in tracking. [7]

2.8 Concluding the tracker research

The various researched tracking methods vary from one to another, with different computational complexities, occlusion handling, reliability/stability and accuracy.

Computational Complexity and Overall Speed

The MOSSE tracker operates well at significantly fps than the other trackers discussed in this report, the KCF tracker operates faster than the MIL tracker.
The TLD tracker is the slowest of the trackers researched here, mainly due to the complexity during runtime. The CSRT works at an average speed, suitable for real-time tracking, the GOTURN tracker is highly subject to the offline dataset given.

Occlusion Handling

The TLD tracker can handle full occlusion for multiple frames and not break lock on target when it reappears. The KCF, MIL and CSRT trackers don't recover from a full occlusion over multiple frames, the CSRT tracker can have its target occluded for longer than the KCF and MIL trackers before being unrecoverable. Based on how closely the offline dataset matches the tracked object, the GOTURN tracker could handle occlusion over multiple frames. Both the MOSSE and MedianFlow trackers perform better with non-occluded objects but have measures to pause and pickup tracking after losing the object to occlusion.

The lack of occlusion handling is a significant drawback for the system being designed, as having the object move behind another object or off-screen would end the location data feed, and therefore force a manual restart of the system.

Reliability, Stability and Accuracy

Both the MIL and CSRT trackers boast a high accuracy and reliability in their respective ideal conditions. The GOTURN tracker is as reliable as the offline dataset is relevant to the object being tracked. MedianFlow has great overall tracking and the algorithm knows when tracking has failed however, the algorithm fails to track objects that change have a large or sudden change of acceleration. The TDL tracker while highly accurate over long periods of time, it's quite unstable due to its high false alarm rate and its tendency to hop between other similar objects temporarily during runtime.

2.9 2-Dimensional Depth Interpretations

An experiment was conducted to gather a deeper understanding of how best to obtain this data with relatively high accuracy using a single camera, while trying to remain somewhat simple for real time calculations.

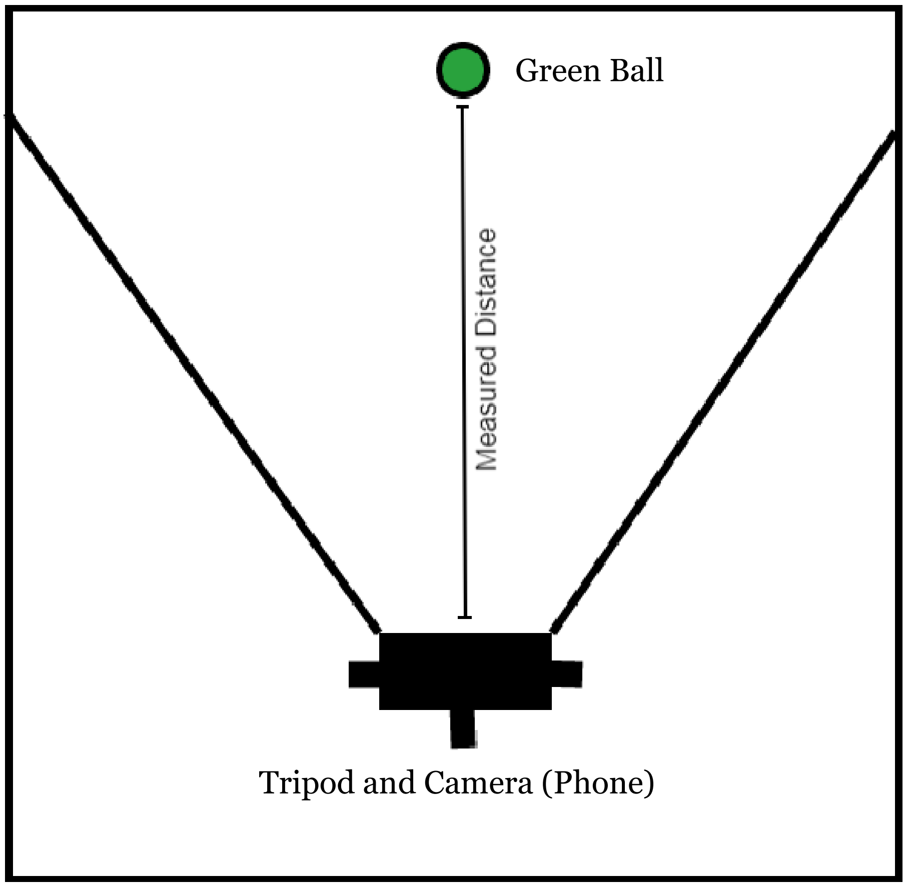


Figure 2-1 - Top-Down view on experiment conducted for depth interpretation analysis.

Using a tripod, tape-measure, green ball and a phone camera in the configuration shown in figure 2-1, the experiment was to take images of the green ball at different distances 25-200cm in intervals of 25 (20cm was done as a hard-minimum value). Distances were measured from the centre of the ball to the centre of the camera with an error of around ±2cm.

After taking these images, the pixels were counted from the horizontal and vertical diameters of the green ball.

The images taken were 4000*3000 pixels in size and the ball had a real-world diameter of 5cm, the camera was also kept at the same height and angle during the experiment.

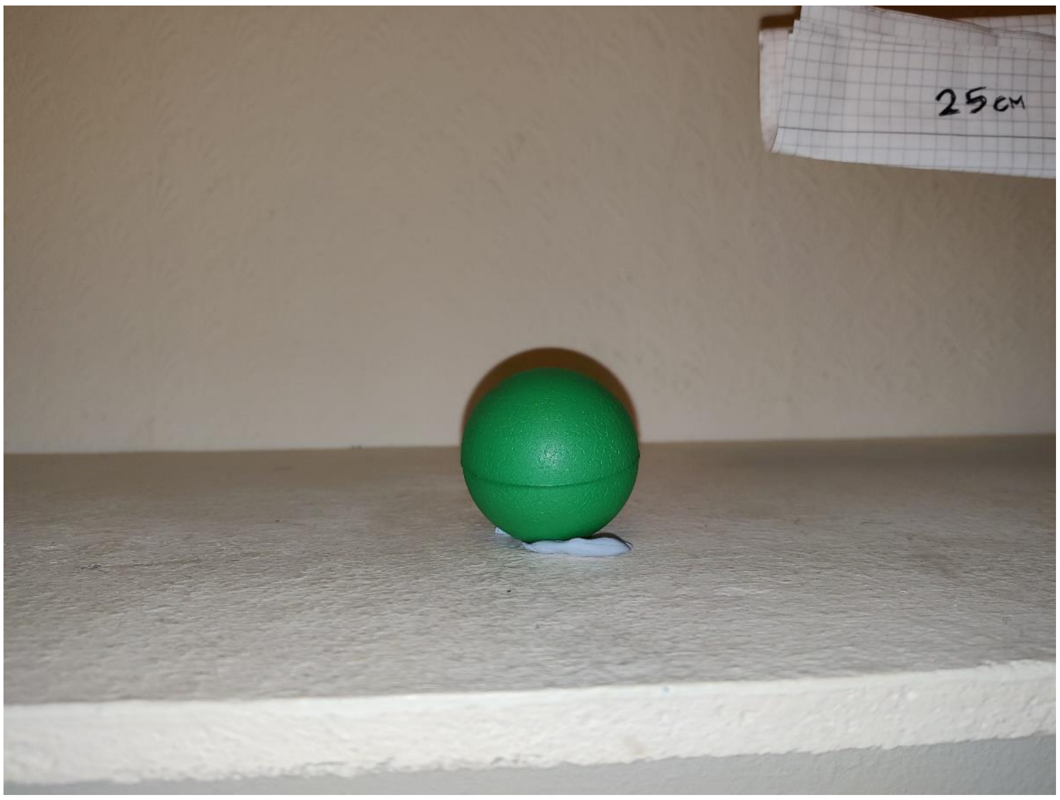
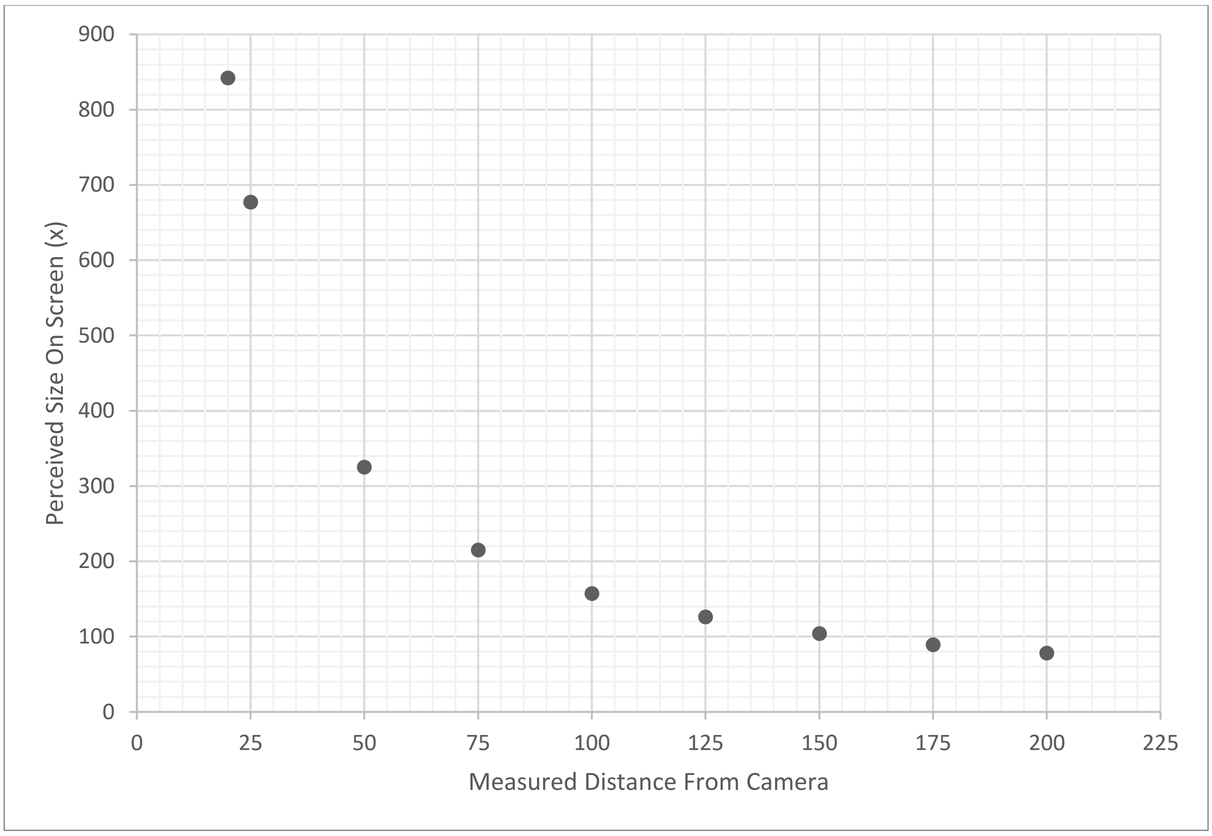


Figure 2-2 - Example image taken from experiment, green ball at 25cm from camera.

(cm)	Perceived Size		% of Screen	
Distance	x	y	x	y
20	842	829	21.050%	27.633%
25	677	674	16.925%	22.467%
50	325	315	8.125%	10.500%
75	215	209	5.375%	6.967%
100	157	150	3.925%	5.000%
125	126	121	3.150%	4.033%
150	104	100	2.600%	3.333%
175	89	84	2.225%	2.800%
200	78	74	1.950%	2.467%

Table 2-1 - Data collected from depth interpretation experiment.

The data shows a rapid increase for the on-screen size of the ball as it gets closer as was expected initially from before the experiment, the perceived size showing a possible exponential trend with a hard upper limit (the size of the screen) and a lower limit of zero.



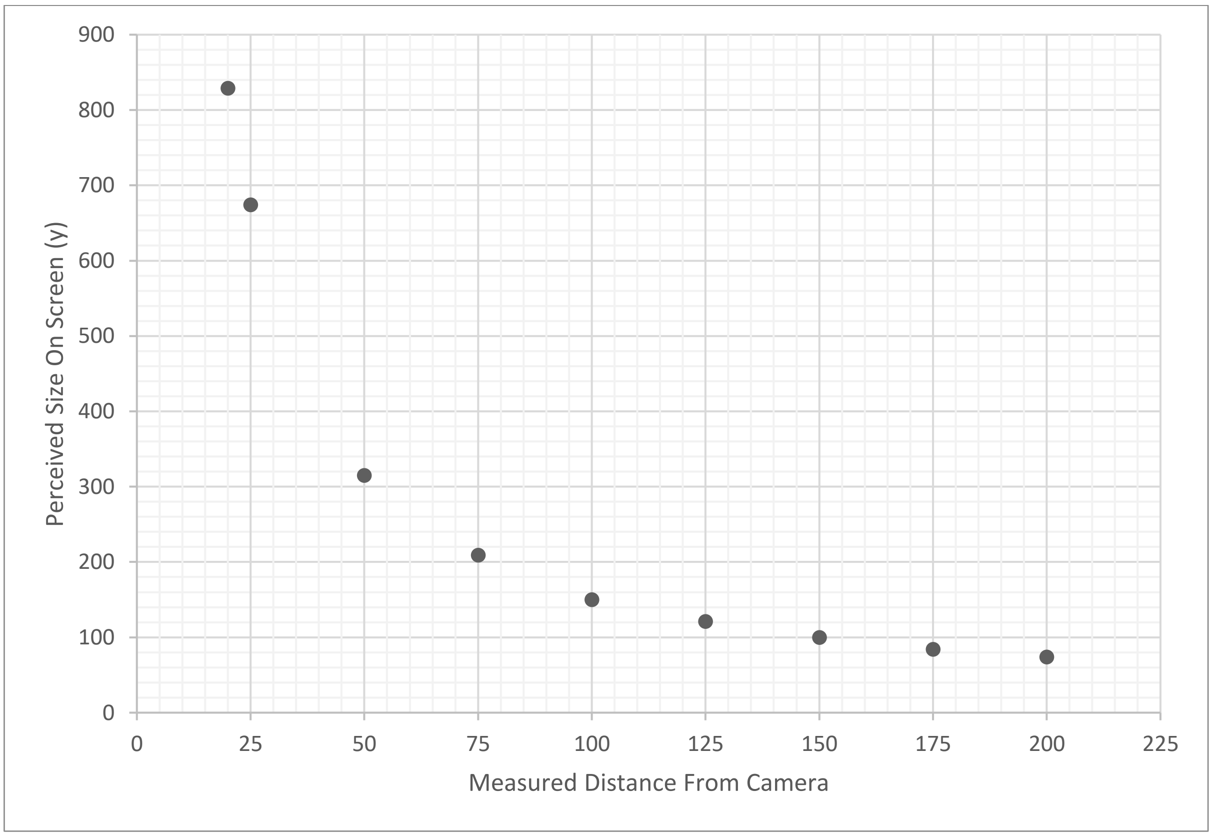


Figure 2-3 - Graphs showing the relationship of the perceived size on screen against the real world measured distance.

(cm)	Pixels per 1cm		% of Screen per Real Diameter		
Distance	x	y	x	y	average
20	168.40	165.80	4.210%	5.527%	4.868%
25	135.40	134.80	3.385%	4.493%	3.939%
50	65.00	63.00	1.625%	2.100%	1.863%
75	43.00	41.80	1.075%	1.393%	1.234%
100	31.40	30.00	0.785%	1.000%	0.893%
125	25.20	24.20	0.630%	0.807%	0.718%
150	20.80	20.00	0.520%	0.667%	0.593%
175	17.80	16.80	0.445%	0.560%	0.503%
200	15.60	14.80	0.390%	0.493%	0.442%

Table 2-2 - Converting pixel into real world distances.

This is the first step in getting location data from these images, by giving pixel values to real-world distances at different depths, the system will be able to figure out where in the object is in a 2-Dimensional plane parallel to the camera lens at any given depth into the 3-Dimensional world.

Triangle Similarity can be used to allow for any distance to be calculated given an initial distance and object width. [8]

$$Perceived\ Focal\ Length = \frac{Object\ Pixel\ Width * Distance}{Object\ Real\ Width}$$

Equation 2-1 - Perceived Focal Length with given real-world data.

$$New\ Distance\ = \frac{Object\ Real\ Width * Perceived\ Focal\ Length}{Object\ Pixel\ Width}$$

Equation 2-2 - Interpreted distance calculated using Perceived Focal Length.

The second equation requires a “perceived focal length” along with the object’s real-world width to calculate new distance values. The perceived focal length can be calculated with one instance of a real-world data being the real distance, width and onscreen width, from this point onwards, new distances can be estimated.

These equations were applied to the data collected, with 100cm being the reference data for the perceived focal length.

(cm)	Calculated Distance			Difference vs Real Distance		
Distance	x	y	average	x	y	average
20	18.646	18.938	18.791	-1.354	-1.062	-1.209
25	23.191	23.294	23.242	-1.809	-1.706	-1.758
50	48.308	49.841	49.063	-1.692	-0.159	-0.938
75	73.023	75.120	74.057	-1.977	0.120	-0.943
100	100.000	104.667	102.280	0.000	4.667	2.280
125	124.603	129.752	127.126	-0.397	4.752	2.126
150	150.962	157.000	153.922	0.962	7.000	3.922
175	176.404	186.905	181.503	1.404	11.905	6.503
200	201.282	212.162	206.579	1.282	12.162	6.579

Table 2-3 - Calculated distances using triangle similarity and the difference between the calculated and real-world data.

With the exception of the y values and therefore the average, the errors don't exceed ±2cm, the errors in the y values mostly come down to part of the ball being occluded and pixel counting in the case of this experiment cannot account for occlusion. However, a few of the tracking algorithms discussed previously have methods of tracking through the occlusion and other methods will be discussed later.

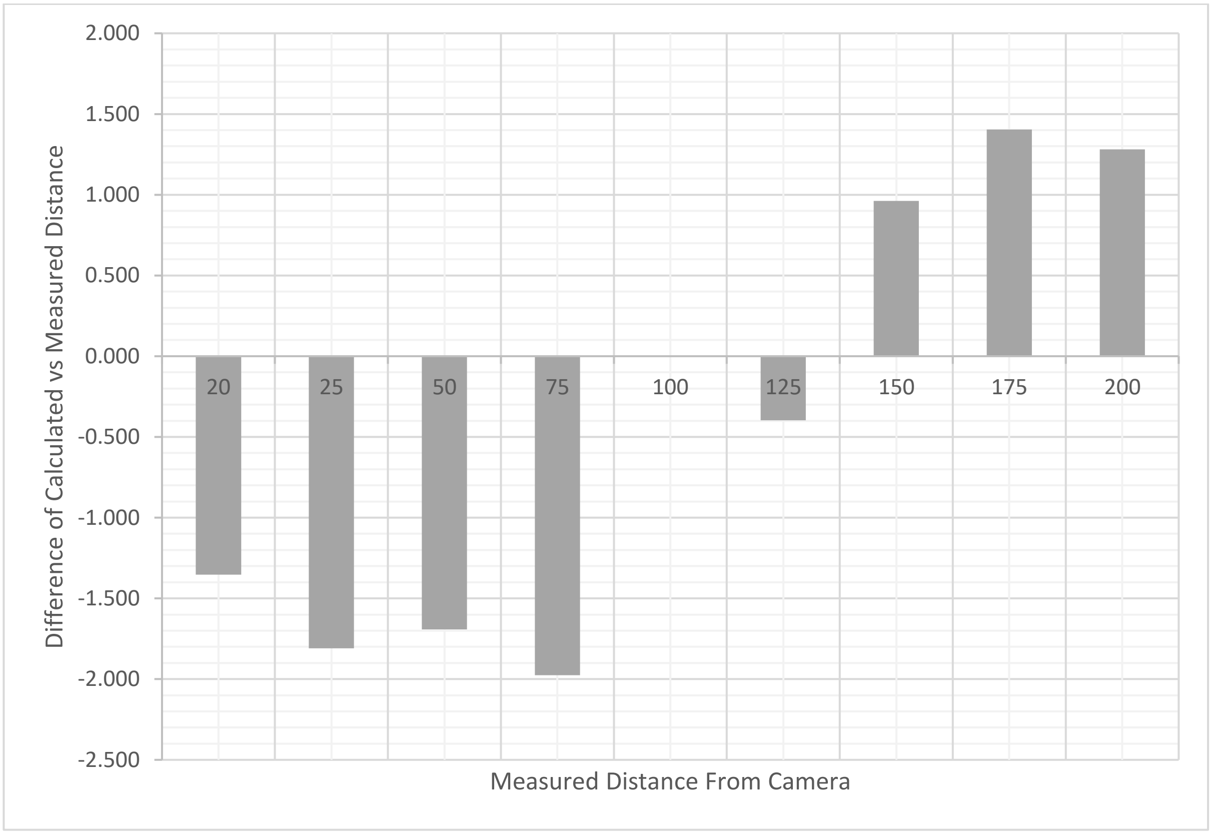


Figure 2-4 - A graph showing the magnitude of error for calculated values compared to real-world values.

There is no error at 100cm due to it being our reference data point. In future testing, different reference data will be compared for performance as the more erroneous the reference data is, the less accurate the calculated distances will be.
Given the errors in measuring the data; ±2cm for real-world distance and ±2.5% for pixel counting, the calculated data is very accurate for the simplicity of performing the calculation.

2.10 Image Thresholding, Binary Masks and Colour Spaces

Image thresholding is a method of segmenting an image, creating a binary mask of the original image to aid with future processing.
These produced masks are used to add exclusion areas to images in which processing is skipped.

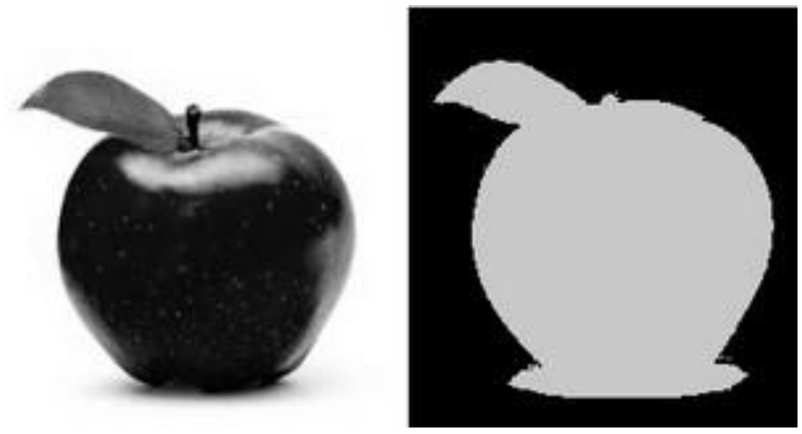


Figure 2-5 - Example of thresholding on grayscale image. [9]

For colour thresholding, an image would first need to be shifted to another colour space as the default RGB space is coded using three colour channels, making it significantly more difficult to create accurate masks based on colour thresholds.
The HSV (Hue, Saturation, Value/Brightness) colour space represents how colours appear under a light source while aligning with a more human perception of colour. The colour space can be represented as a cylindrical model, with Hue representing the angular dimension of the cylinder, Saturation represented as the distance from centre and Value/Brightness represented as the height. The primary colours are present at 0°, 120° and 240° for red, green and blue, respectively. This representation of colour is coded in a single channel and therefore is much easier to work with for image processing.

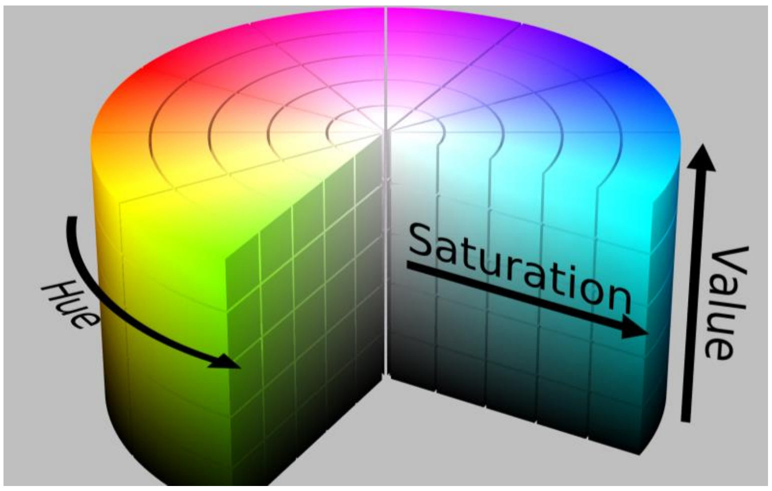


Figure 2-6 - Cylindrical representation of the HSV colour space

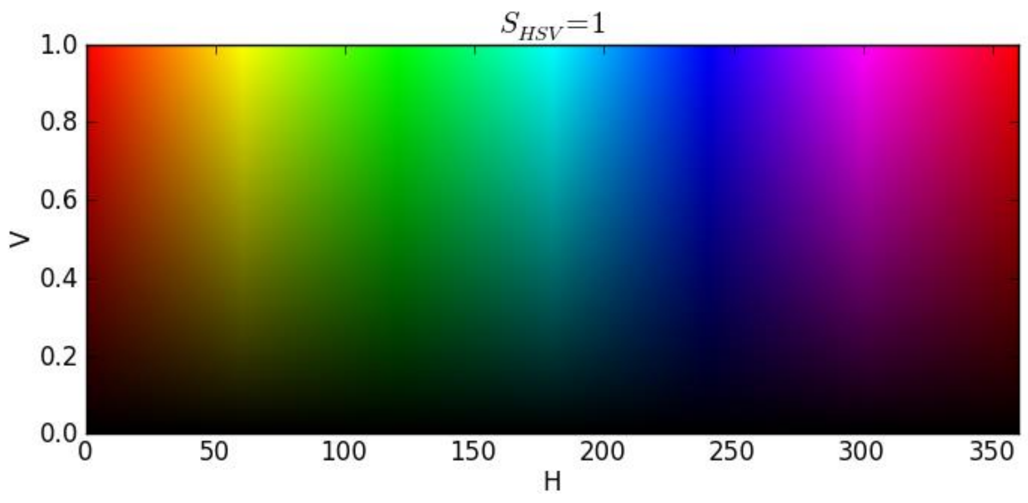


Figure 2-7 - 2-Dimensional graph of the HSV colour space with a constant saturation value of 1.

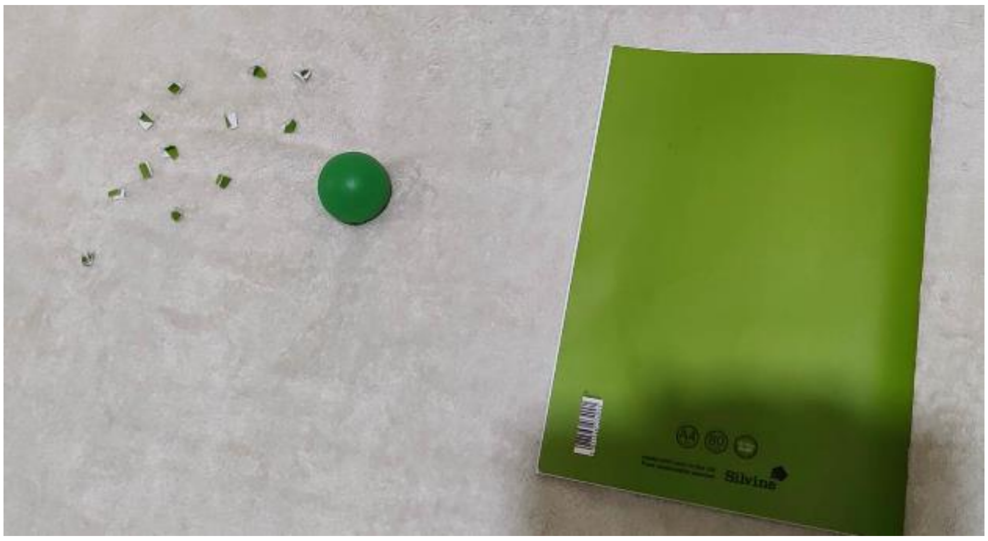


Figure 2-8 - Image of artificial noise, a green ball and a large green object.

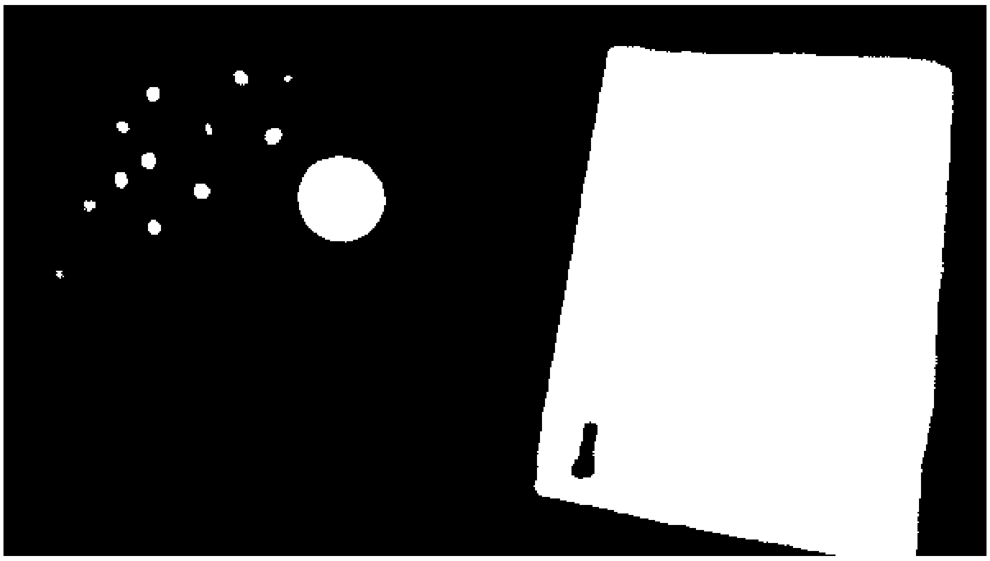


Figure 2-9 - Binary Mask of Figure 2-7, using OpenCV Python.

2.11 Sound Localisation

Several cues are used by the auditory system for sound source localisation such as, time difference, intensity differences and possible repeats/echoes.

A 3-Dimensional representation of sound can be described using; the azimuth (horizontal angle), the elevation (vertical angle) and the distance.

A change in azimuth or elevation can be simulated with a delay between the right and left audio channels, causing an ITD (Interaural time difference). This Interaural time difference is one of the cues in which a listener can accurately pinpoint a direction for an audio source.
Altering the amplitude can simulate a change in distance, with adding delayed reverberated signals simulating audio reflecting off surfaces in the distance.
At much further distances, higher frequencies dissipate completely, leaving only low-end muffled bass.

3. Development

3.1 Introduction to Libraries and Initialisation Code

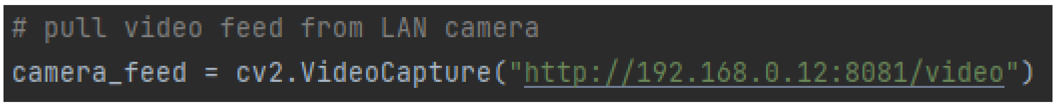
The program will be using:

- IP Webcam Application [10]
- OpenCV Python library 4.5.2.52 [11]
- Imutils 0.5.4 [12]
- Numpy 1.20.2 [13]
- Pyglet 1.5.16 [14]
- OpenAL 0.4.0 [15]
- Ffmpeg [16]

Written in python 3.9 [17] using PyCharm Community Edition [18].

The OpenCV Python library allows for advanced tracking methods and image manipulation and will be the main library that the system will refer to.
A request is made at http://AAA.BBB.CCC.DDD:EEEE/video and bound to a VideoCapture object (provided by OpenCV). The video is provided by the IP Webcam application over LAN from a mobile device, allowing that device to function as a wireless webcam for the program to use.

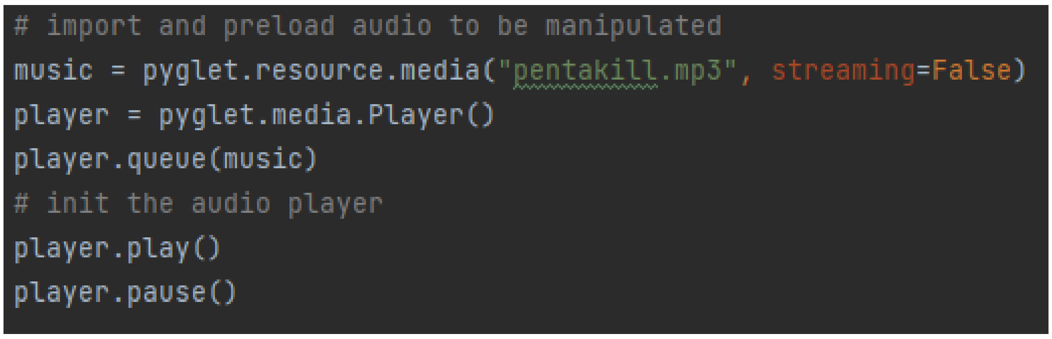
The video is stored as collection of frames for which each frame can be read as a multi-dimensional array.



```
# pull video feed from LAN camera
camera_feed = cv2.VideoCapture("http://192.168.0.12:8081/video")
```

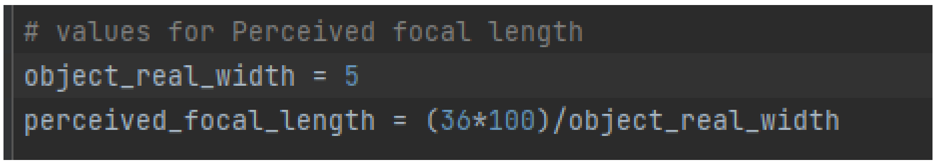
Figure 3-1 - Code for basic video retrieval from LAN in Python.

The Pyglet library is primarily for game development, it is however being used for its audio handling capabilities with real time audio manipulation. It is paired up with the openAL drivers and ffmpeg to handle the loading, playback and movement of audio.



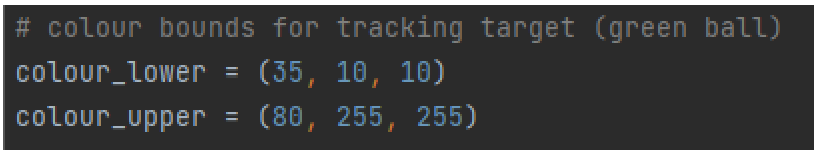
```
# import and preload audio to be manipulated
music = pyglet.resource.media("pentakill.mp3", streaming=False)
player = pyglet.media.Player()
player.queue(music)
# init the audio player
player.play()
player.pause()
```

Figure 3-2 - Setting up audio queue and initialising the player.



```
# values for Perceived focal length
object_real_width = 5
perceived_focal_length = (36*100)/object_real_width
```

Figure 3-3 - Data used from real world measurements and experimental data gathered in section 2.9.



```
# colour bounds for tracking target (green ball)
colour_lower = (35, 10, 10)
colour_upper = (80, 255, 255)
```

Figure 3-4 - HSV colour space ranges chosen based on experimenting with green ball.

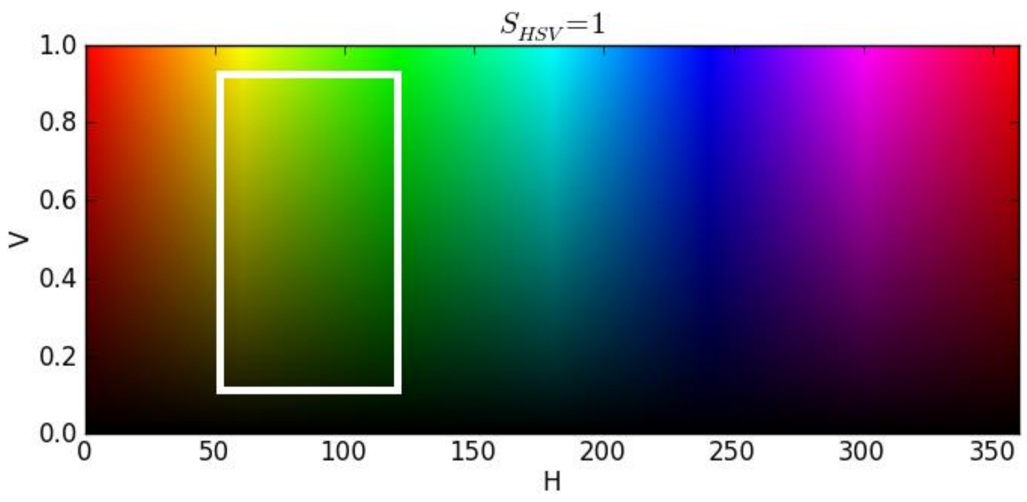
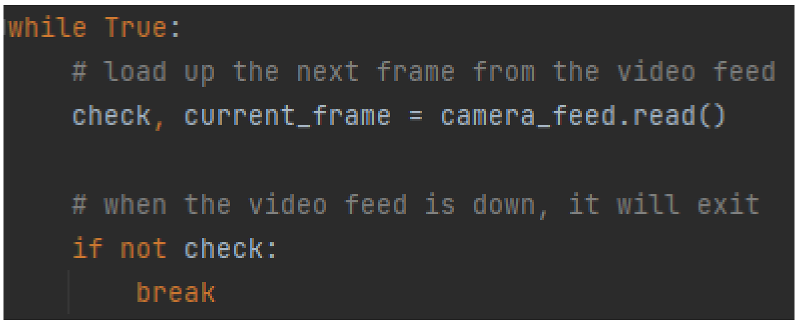


Figure 3-5 - Visual annotation of chosen range on 2-Dimensional HSV colour space graph

3.2 Main loop basics

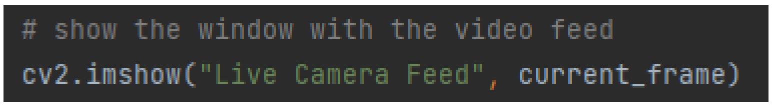
The main program runs on a while loop, broken out of only on one condition: the video feed going down.



```
while True:
    # load up the next frame from the video feed
    check, current_frame = camera_feed.read()

    # when the video feed is down, it will exit
    if not check:
        break
```

Figure 3-6 - Constantly reads the next frame in, the "check" is returned true on successful read.



```
# show the window with the video feed
cv2.imshow("Live Camera Feed", current_frame)
```

Figure 3-7 - OpenCV window creation, feeding in the frame that were read earlier in the loop.

From these lines of code, the result is a window called "Live Camera Feed" that shows exactly what the mobile device's camera is currently looking at, as if it were a wireless camera.

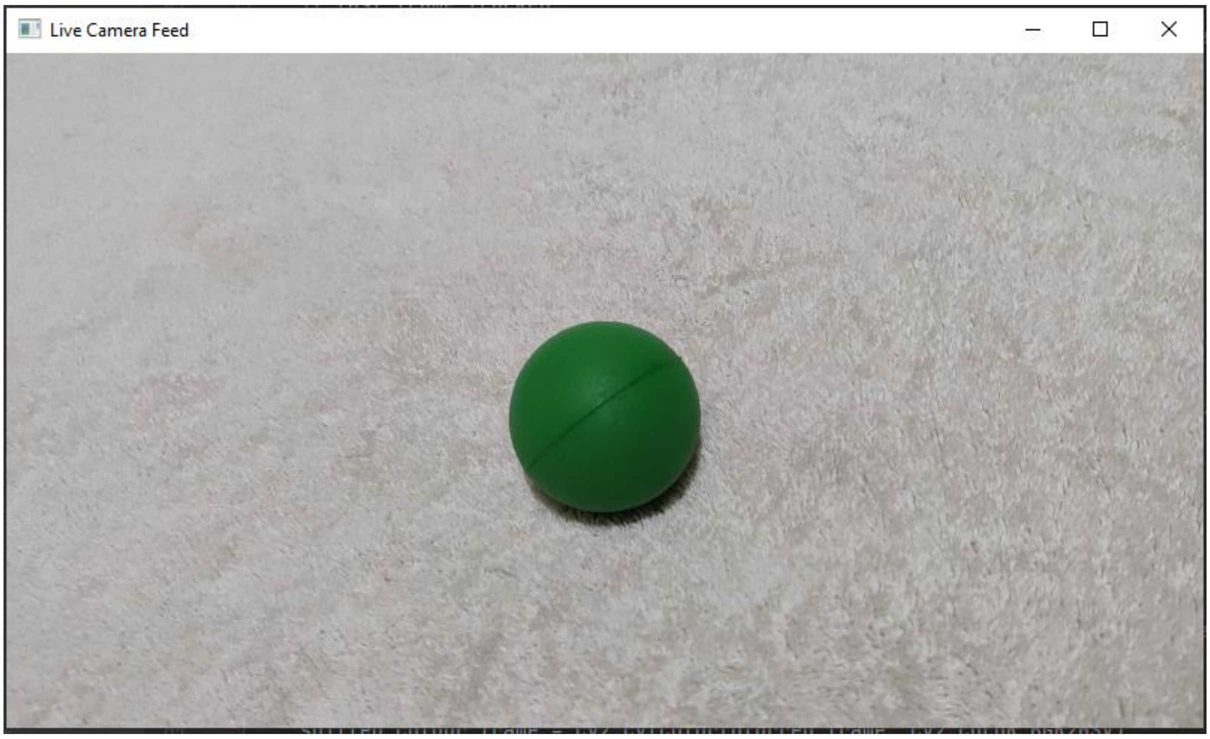
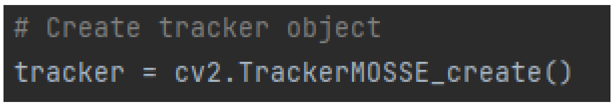


Figure 3-8 - "Live Camera Feed" window with the green ball to be tracked.

3.3 Specific tracker setup

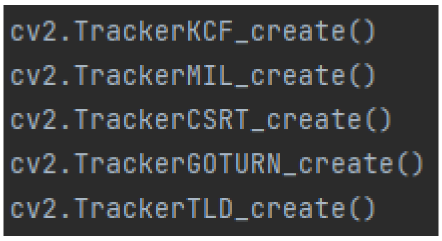
The OpenCV-Python library offers a range of different trackers, all the trackers discussed in section 2 are available to use.



```
# Create tracker object
tracker = cv2.TrackerMOSSE_create()
```

Figure 3-9 - Creation of tracker object using OpenCV.

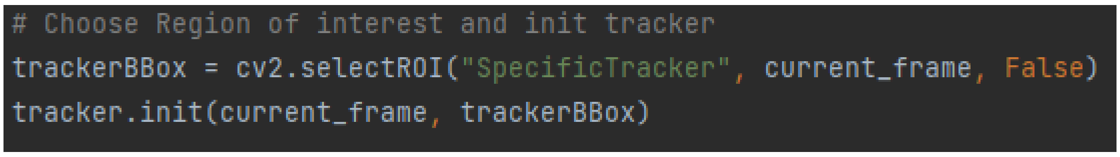
The tracker can also be substituted for:



```
cv2.TrackerKCF_create()
cv2.TrackerMIL_create()
cv2.TrackerCSRT_create()
cv2.TrackerGOTURN_create()
cv2.TrackerTLD_create()
```

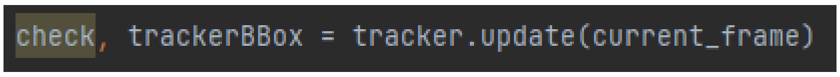
Figure 3-10 - Various trackers available with OpenCV.

A region of interest is initially selected, this is what the tracker will use as its initial position and will attempt to track it.



```
# Choose Region of interest and init tracker
trackerBBox = cv2.selectROI("SpecificTracker", current_frame, False)
tracker.init(current_frame, trackerBBox)
```

Figure 3-11 - Region of interest selection, followed by initialisation of tracker.



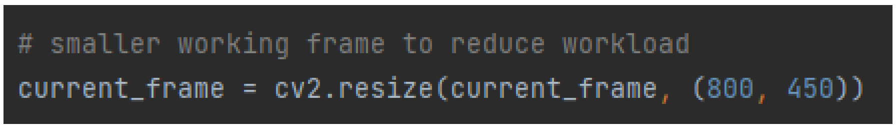
```
check, trackerBBox = tracker.update(current_frame)
```

Figure 3-12 - Called to update the tracker with a new position.

3.4 Specific tracker issues

While these trackers did work, although some worked significantly better than others, the MIL, CSRT, MEDIANFLOW and TLD trackers operated at an average frame rate of about 5hz. The GOTURN tracker has the same low frame issue, but also requires a model to run off.
These trackers are simply unusable in this context due to their low operating speeds. Both the MOSSE and KCF trackers work well over 60hz however, these trackers tend to drift off target especially when the object is moving at high speeds. The main issue with the trackers in OpenCV is the lack of fluidity, the trackers output coordinates with great fluctuations, which did not work too well with the openAL drivers. Tracking contours on a binary mask worked exceedingly well, of which will be discussed in the next section.

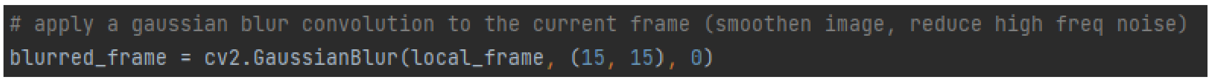
3.5 Thresholding, binary masking and noise reduction



```python
# smaller working frame to reduce workload
current_frame = cv2.resize(current_frame, (800, 450))
```

Figure 3-13 - Compressing the working frame to reduce the workload.

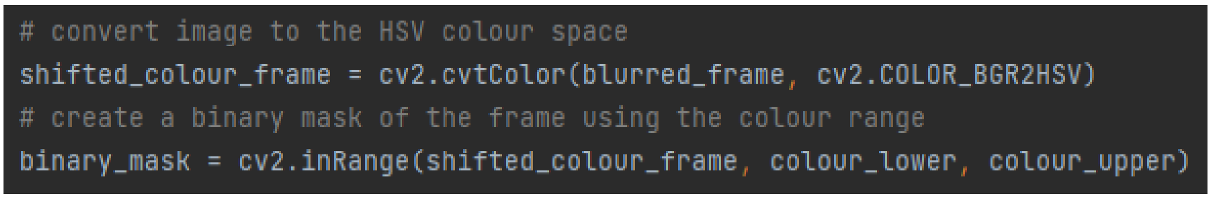
To begin working on the frame, it needs to be prepared for processing. Firstly, the frame is resized to be a smaller overall workload (although it is capable of FHD 60hz without any issues), the frame is reduced to 41% of its original size.



```python
# apply a gaussian blur convolution to the current frame (smoothen image, reduce high freq noise)
blurred_frame = cv2.GaussianBlur(local_frame, (15, 15), 0)
```

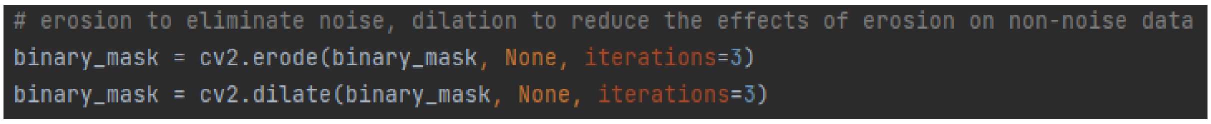
Figure 3-14 - Gaussian blurring for high frequency noise reduction.

Three methods are used to reduce noise in our image.
A Gaussian blur is applied before thresholding, reducing high frequency noise by smoothening and blending the colours in the image without causing too much collateral visual damage to the larger objects being tracked.



```python
# convert image to the HSV colour space
shifted_colour_frame = cv2.cvtColor(blurred_frame, cv2.COLOR_BGR2HSV)
# create a binary mask of the frame using the colour range
binary_mask = cv2.inRange(shifted_colour_frame, colour_lower, colour_upper)
```

Figure 3-15 - HSV colour space shifting, followed by thresholding of the image with the given ranges at start up.



```python
# erosion to eliminate noise, dilation to reduce the effects of erosion on non-noise data
binary_mask = cv2.erode(binary_mask, None, iterations=3)
binary_mask = cv2.dilate(binary_mask, None, iterations=3)
```

Figure 3-16 - Triple iteration Erosion followed by dilation.

The second method in noise reduction is erosion-dilation, every contour has its edges eroded away and dilated back to its original size. This has the effect of causing unwanted small objects in the mask to completely disappear, while causing minimal collateral visible damage to the remaining mask.

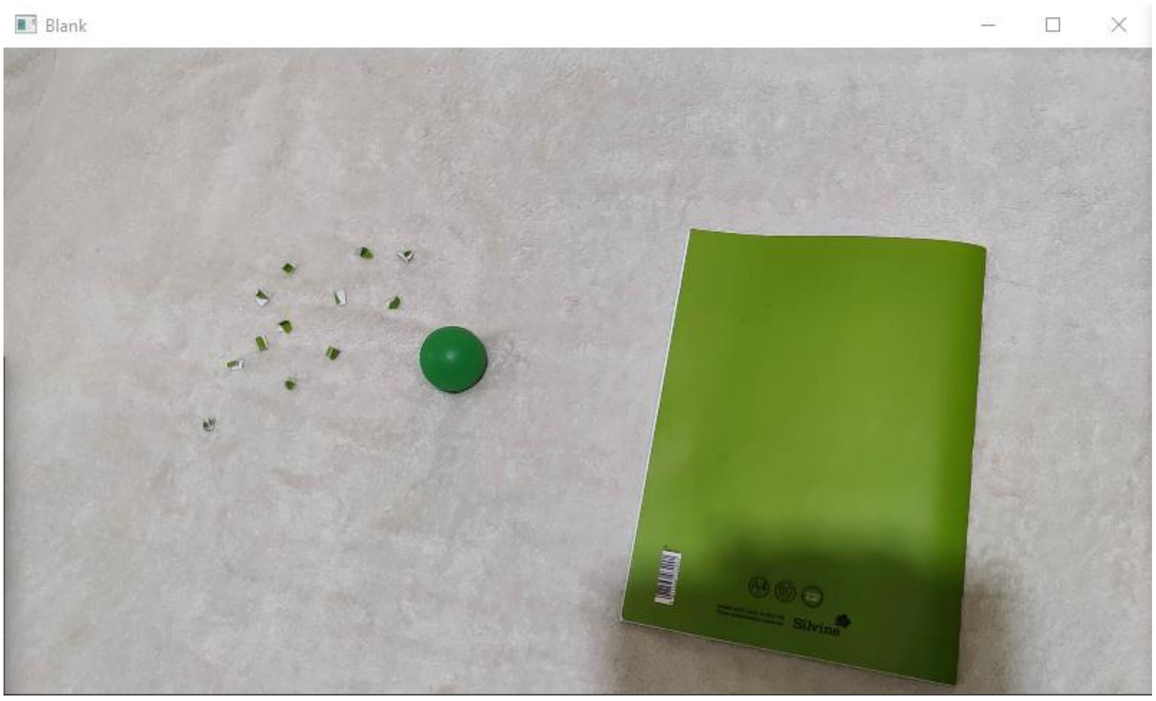


Figure 3-17 - Noise reduction example (original image).

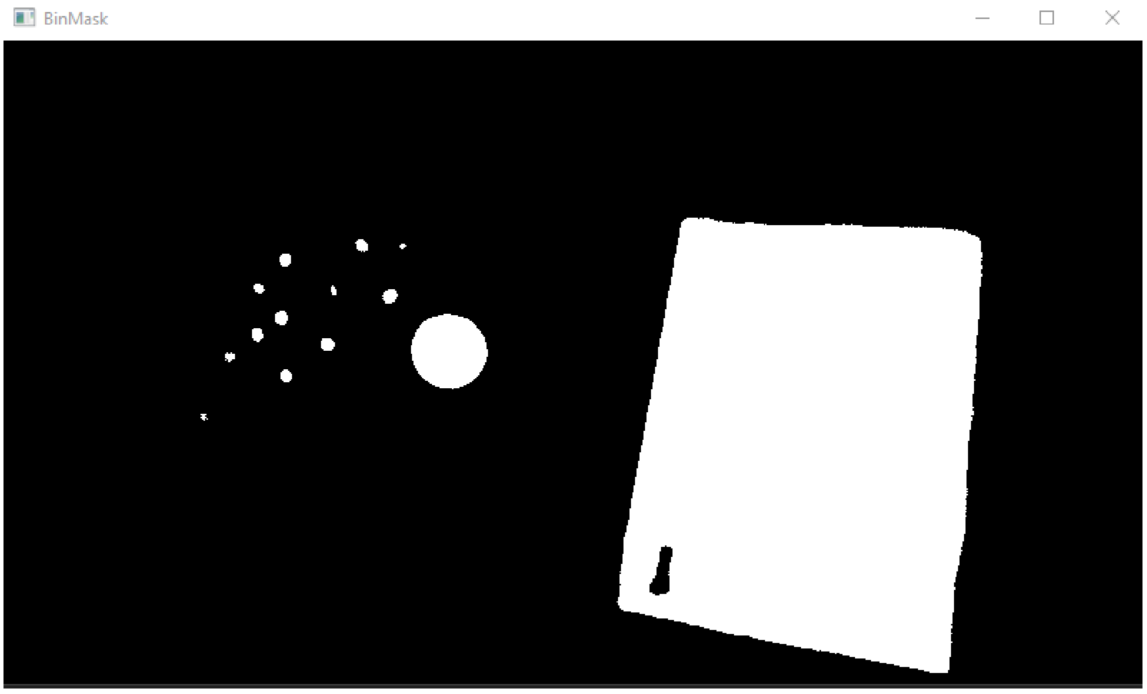


Figure 3-18 - Noise reduction example (binary mask).

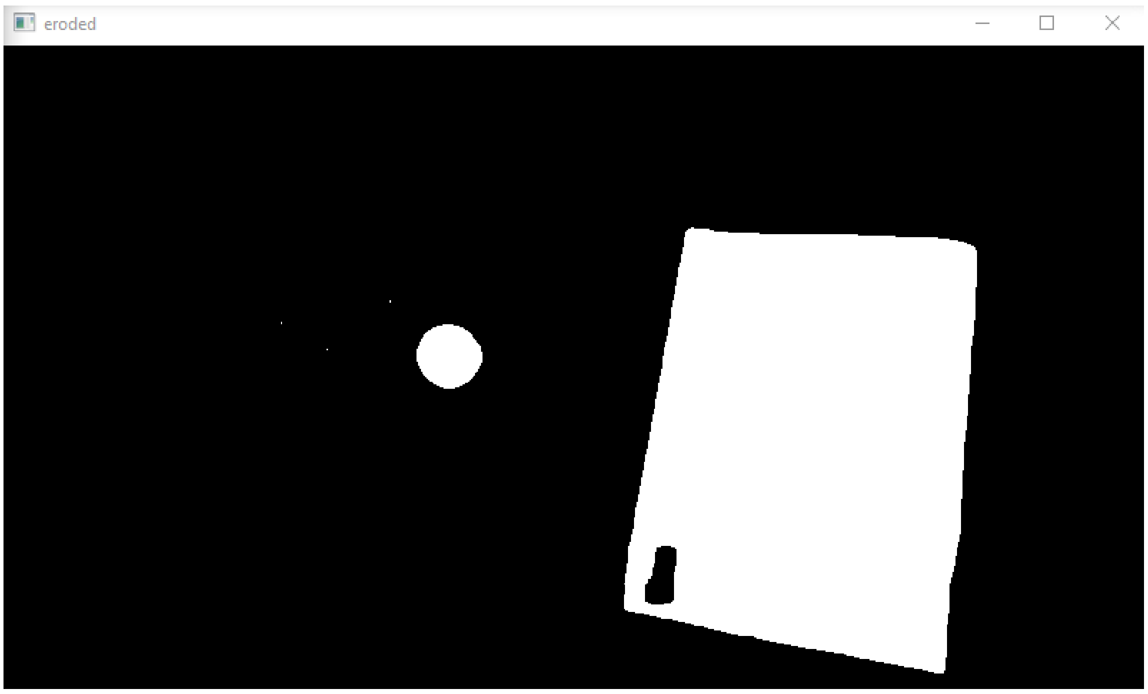


Figure 3-19 - Noise reduction example (triple erosion).

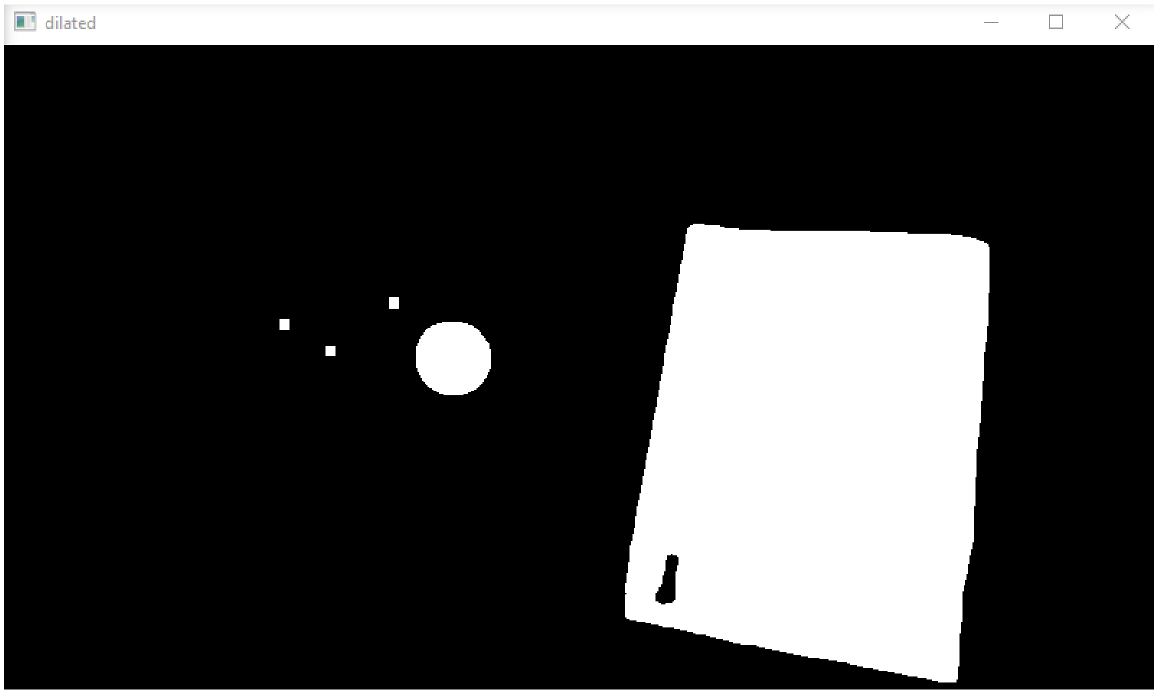
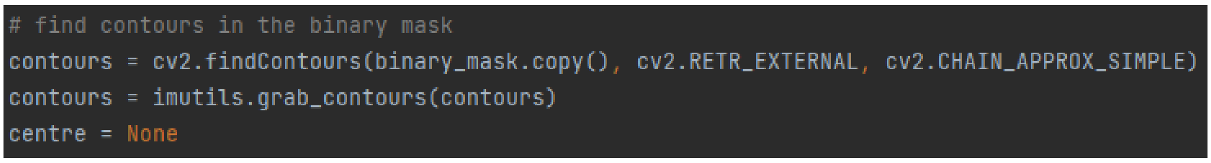


Figure 3-20 - Noise reduction example (triple dilation).

From figure 3-17 to 3-20 the effectiveness of the triple erosion-dilation method, removing 75% of all the artificial noise objects in the original mask.

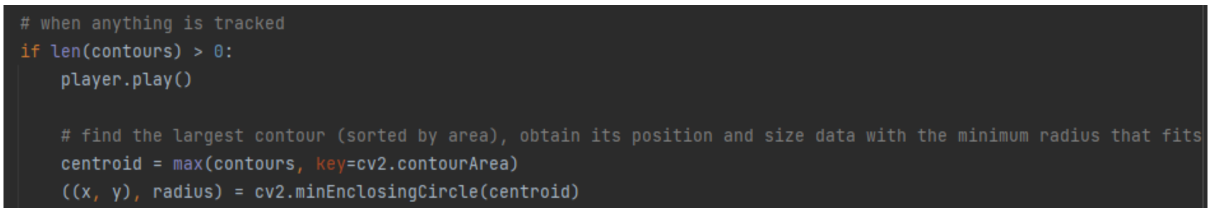
3.6 Contours, centroids and line of sight audio



```
# find contours in the binary mask
contours = cv2.findContours(binary_mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
contours = imutils.grab_contours(contours)
centre = None
```

Figure 3-21 - Gathering a list of all the contours in the image mask.

A list of contour objects is made from the binary image, each with varying size and position.

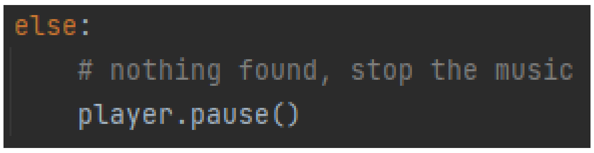


```
# when anything is tracked
if len(contours) > 0:
    player.play()

    # find the largest contour (sorted by area), obtain its position and size data with the minimum radius that fits
    centroid = max(contours, key=cv2.contourArea)
    ((x, y), radius) = cv2.minEnclosingCircle(centroid)
```

Figure 3-22 - Music resumes and the largest contour's centroid is calculated.

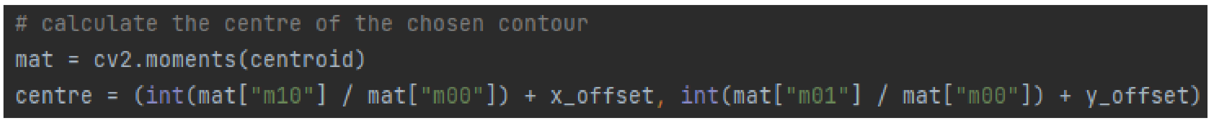
If there is more than one contour found in the mask, the player resumes music for the duration of the frame, the largest area contour is selected and the weighted average of all the pixels making up the contour is calculated (centroid).
The smallest circle that can enclose the object is calculated, giving an x, y and radius value in pixels.



```
else:
    # nothing found, stop the music
    player.pause()
```

Figure 3-23 - Line of sight requirement given to the audio, pausing it when no contours are observed in the mask.

Music is paused when there is no object in view, giving the effect of a "line of sight" requirement for the audio which raises possible applications of this kind of interaction, possibly to help with accessibility, blindness and other limitations.



```
# calculate the centre of the chosen contour
mat = cv2.moments(centroid)
centre = (int(mat["m10"] / mat["m00"]) + x_offset, int(mat["m01"] / mat["m00"]) + y_offset)
```

Figure 3-24 - Centre of the centroid is calculated.

The weighted average of object pixel intensities (image moment) is calculated, this gives the centre of the enclosing circle.

```
# draw a circle enclosing the contour with a dot in the centre
cv2.circle(current_frame, (int(x), int(y)), int(radius), (0, 255, 255), 2)
cv2.circle(current_frame, centre, 5, (0, 0, 255), -1)
```

Figure 3-25 - Circle drawn, representing the tracking data.

A circle is drawn onto the frame to visually represent the success of the tracker and where the program thinks the object currently is.

```
# apply formula to calculate distance given the real world parameters of the tracked object
distance = (object_real_width * perceived_focal_length) / (radius * 2)
# frame centre to tracking centre line
cv2.line(current_frame, (400, 225), (math.floor(x), math.floor(y)), (0, 0, 0), 4)
```

Figure 3-26 - Applying equation 2-2 to calculate the distance of the object given real world parameters and onscreen size.

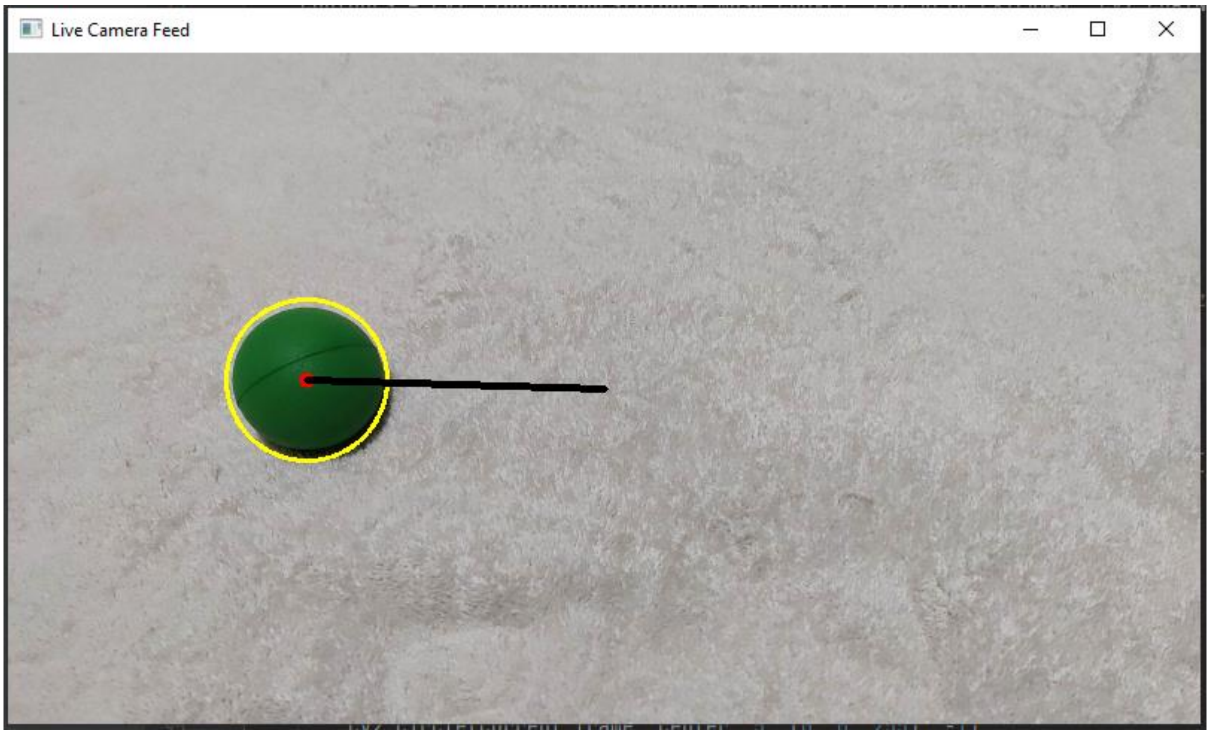


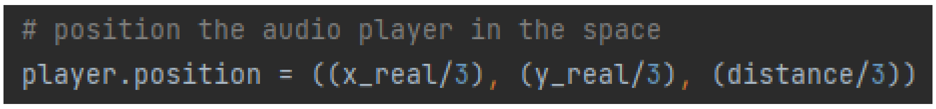
Figure 3-27 - Example showing the tracking accuracy on a still object.

The tracking works well with these basic methods however, there are some major issues to be discussed in section 3.7.

```
# data conversion from pixels to cm
pixel_ratio = 5.0 / (2.0 * radius)
x_real = (x - 400.0) * pixel_ratio
y_real = (225 - y) * pixel_ratio
```

Figure 3-28 - Pixels to centimetres conversion.

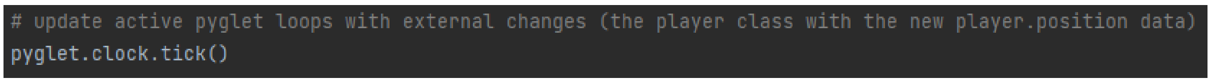
Knowing how much a pixel is worth in real world units, allows the program to project the display to the parallel plane the object occupies. The x-y values can be calculated, and therefore the displacement from the camera can be calculated too.
The centre of the screen is considered (0,0), therefore the object could be referenced to using a polar coordinate system.



```python
# position the audio player in the space
player.position = ((x_real/3), (y_real/3), (distance/3))
```

Figure 3-29 - Mapping object position in 3-dimensional space to the player object.

The player is moved to the same location as the object virtually causing the illusion of the object emitting the audio. Only 33% of the x-y position was chosen to contribute to the player position after extensive testing to match the audio location with what the brain would expect if the ball were emitting the sound.



```python
# update active pyglet loops with external changes (the player class with the new player.position data)
pyglet.clock.tick()
```

Figure 3-30 - Ticking the pyglet loop forwards, allowing for updates to take effect.

3.7 Local area frame tracking

A major issue with the tracking success of the program was that any time a larger object that falls within the colour range defined comes into frame, the tracker will immediately begin following that object instead.
It is programmed to take the largest contour in the scene and assume that is the object of interest to be tracked.

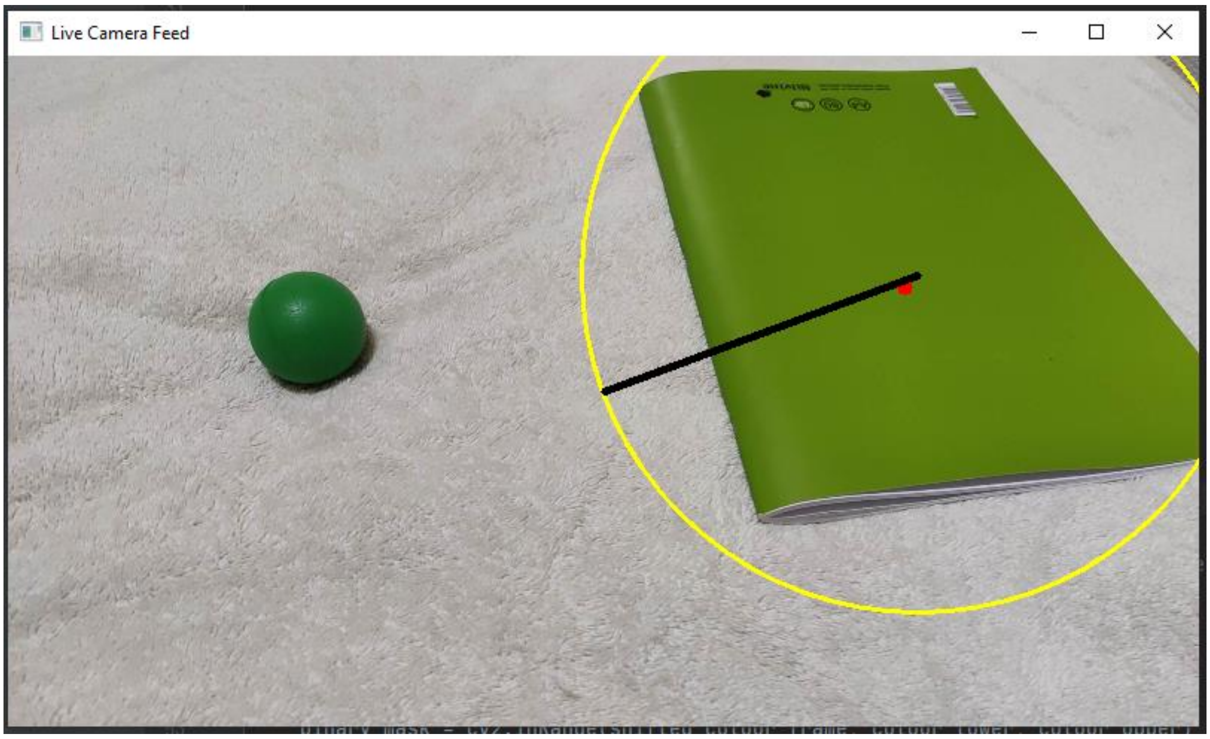
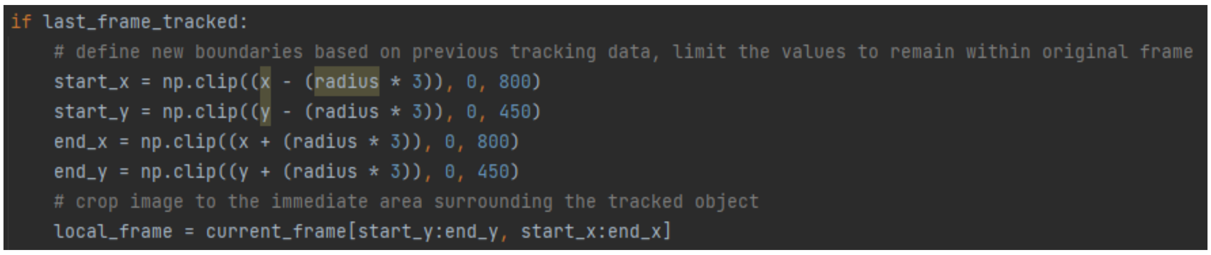


Figure 3-31 - Object size priority, green ball not being tracked.

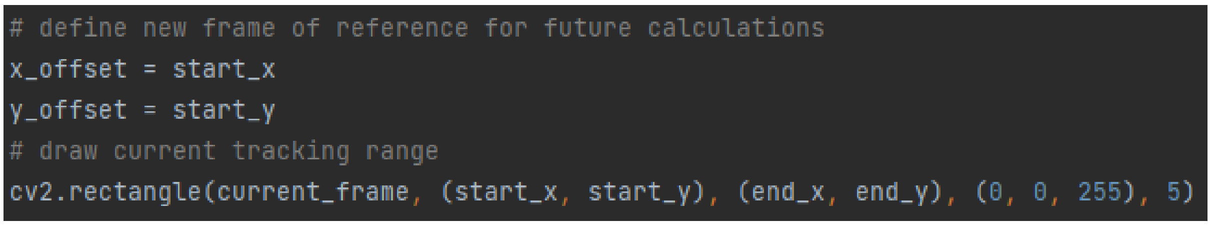
An easy solution to this is to limit the tracking area.
By using previous frame tracking data, the program is able to section off a small area around the tracked object and search only that region for contours, significantly increasing the chance that the object isn't lost to a larger object in the greater frame.



```
if last_frame_tracked:
    # define new boundaries based on previous tracking data, limit the values to remain within original frame
    start_x = np.clip((x - (radius * 3)), 0, 800)
    start_y = np.clip((y - (radius * 3)), 0, 450)
    end_x = np.clip((x + (radius * 3)), 0, 800)
    end_y = np.clip((y + (radius * 3)), 0, 450)
    # crop image to the immediate area surrounding the tracked object
    local_frame = current_frame[start_y:end_y, start_x:end_x]
```

Figure 3-32 - Local area frame range definition.

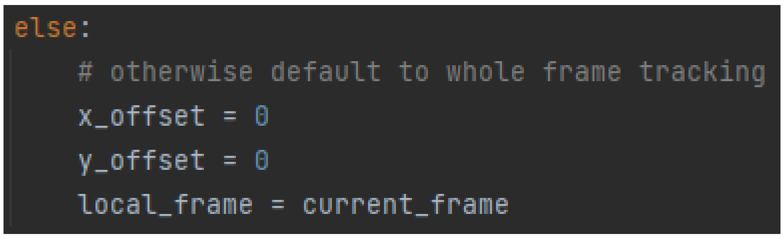
Using a segment of the image to do tracking in requires the program to account for the new frame of reference, a simple x-y offset takes care of value correction for future calculations.



```
# define new frame of reference for future calculations
x_offset = start_x
y_offset = start_y
# draw current tracking range
cv2.rectangle(current_frame, (start_x, start_y), (end_x, end_y), (0, 0, 255), 5)
```

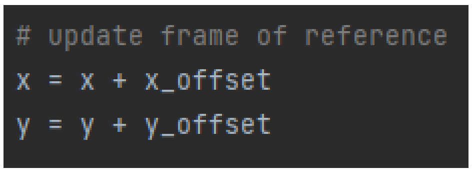
Figure 3-33 - Create offset to correct for new frame of reference.

In the case the object was lost in the previous frame, there needs to be a fallback as there is no way to guarantee the object will return into frame in the same location it was lost in.
There are improvements to offscreen and occlusion, these are discussed in section 3.8.



```
else:
    # otherwise default to whole frame tracking
    x_offset = 0
    y_offset = 0
    local_frame = current_frame
```

Figure 3-34 - Normal tracking when last frame had no tracking data.



```
# update frame of reference
x = x + x_offset
y = y + y_offset
```

Figure 3-35 - Shift the working frame of reference to account for the local area frame of reference.

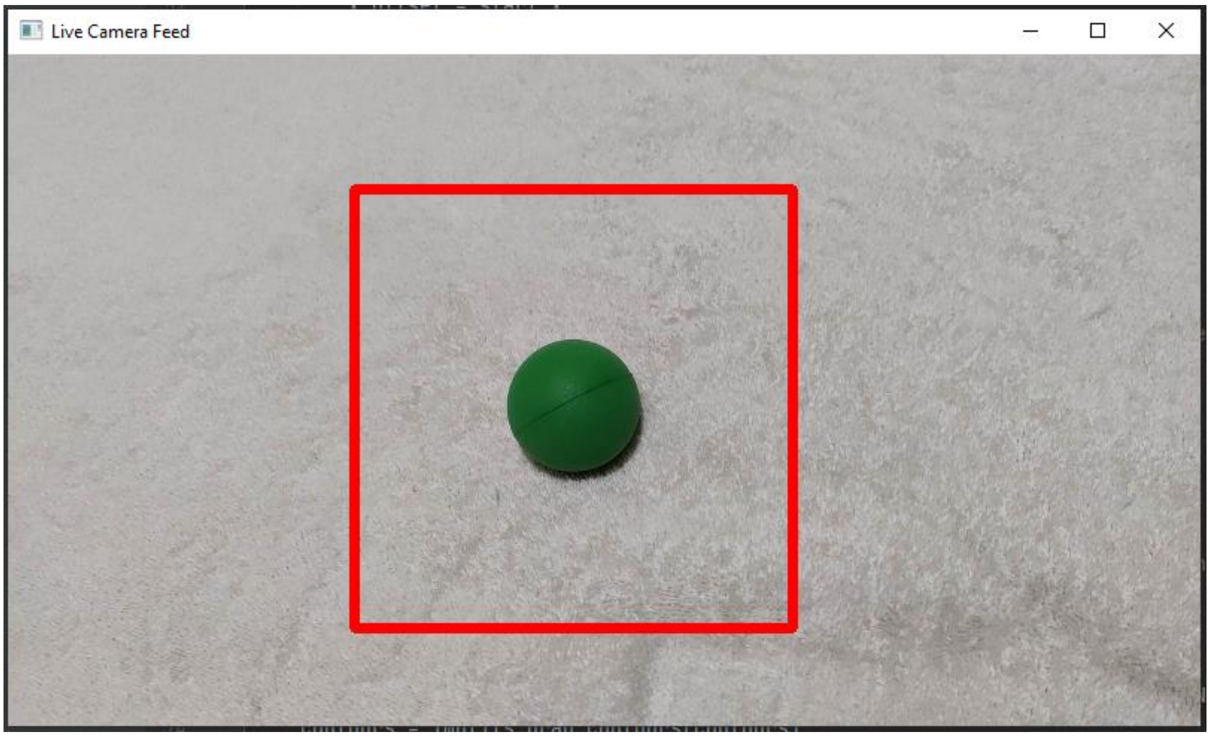
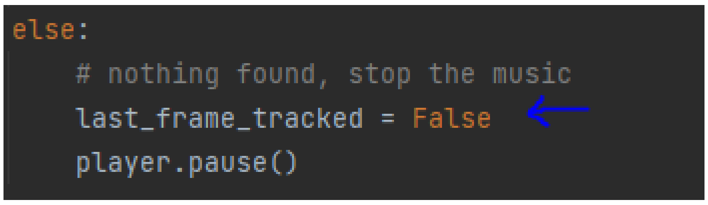


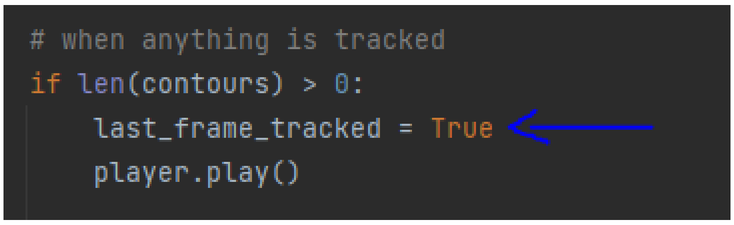
Figure 3-36 - Example, red box shows new tracking area.

Previous code had a couple additions to toggle the new tracking area on and off.



```
else:
    # nothing found, stop the music
    last_frame_tracked = False
    player.pause()
```

Figure 3-37 - Disable local area frame tracking when no previous data.



```
# when anything is tracked
if len(contours) > 0:
    last_frame_tracked = True
    player.play()
```

Figure 3-38 - Enable local area frame tracking for the next frame when there currently is tracking data.

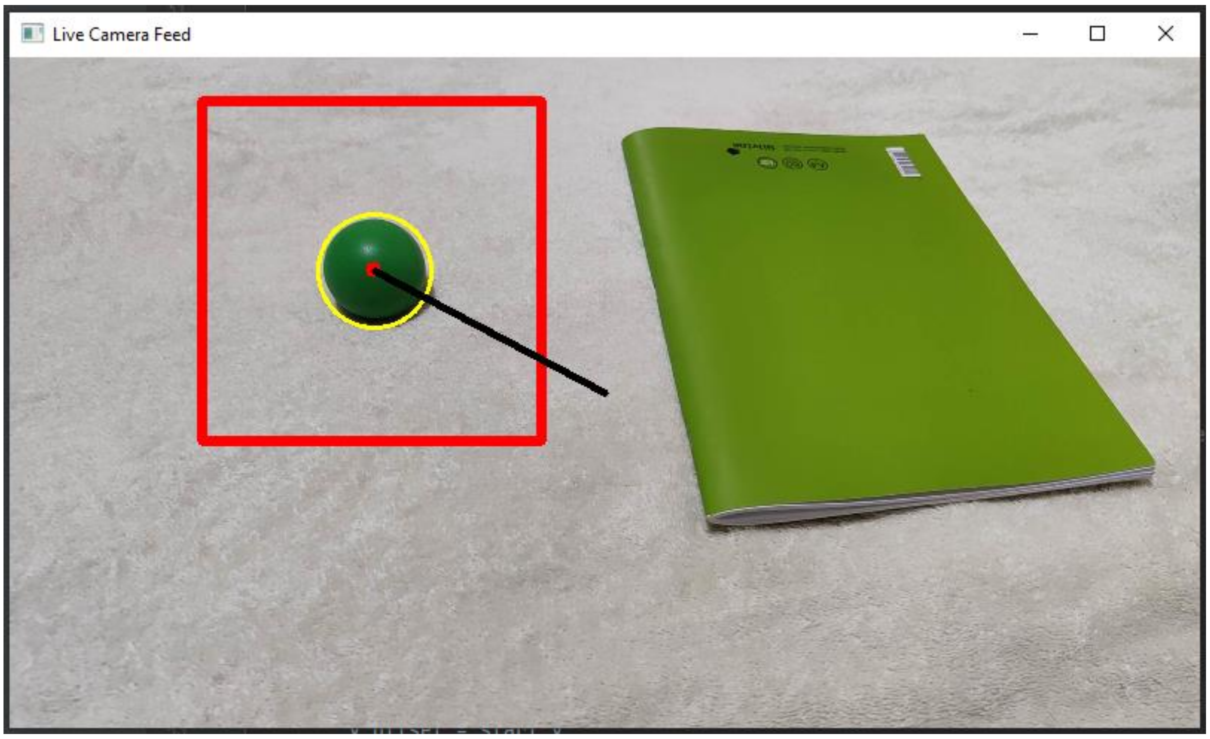


Figure 3-39 - Tracking maintains lock on initial object with local area frame tracking.

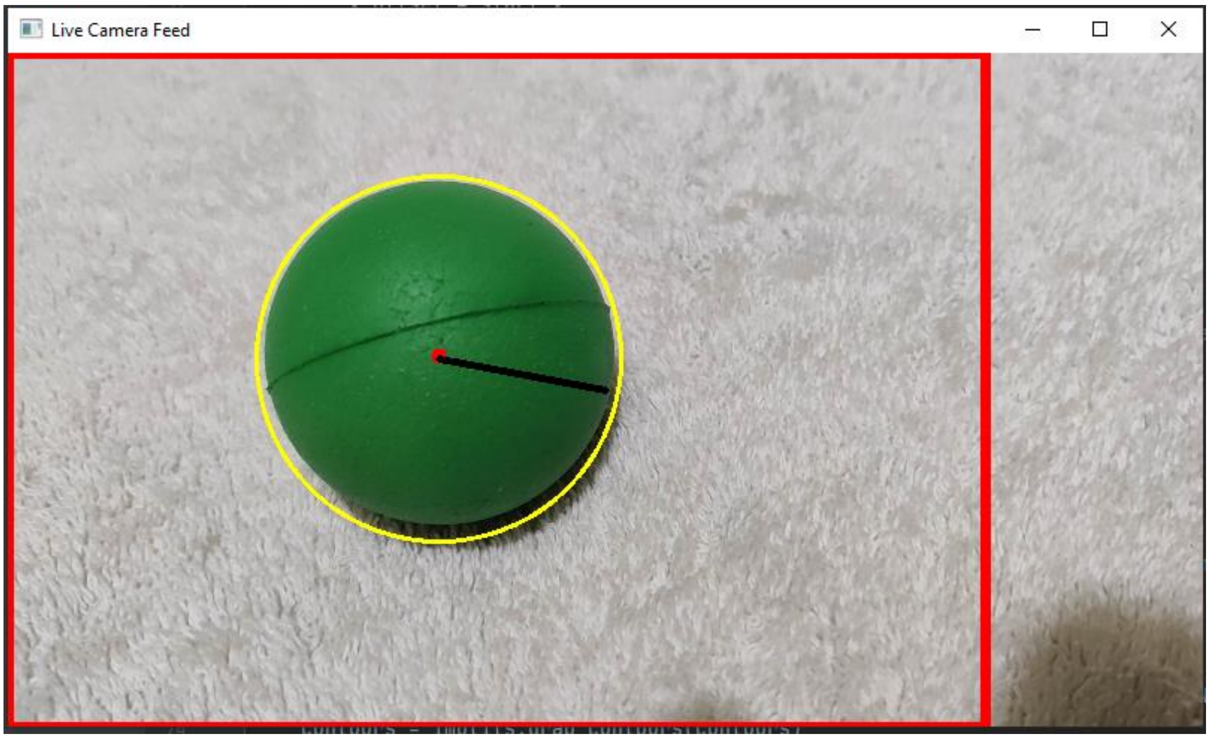


Figure 3-40 - Local area frame is bound by its parent frame.

The local area frame cannot exceed the boundaries set by the whole frame.

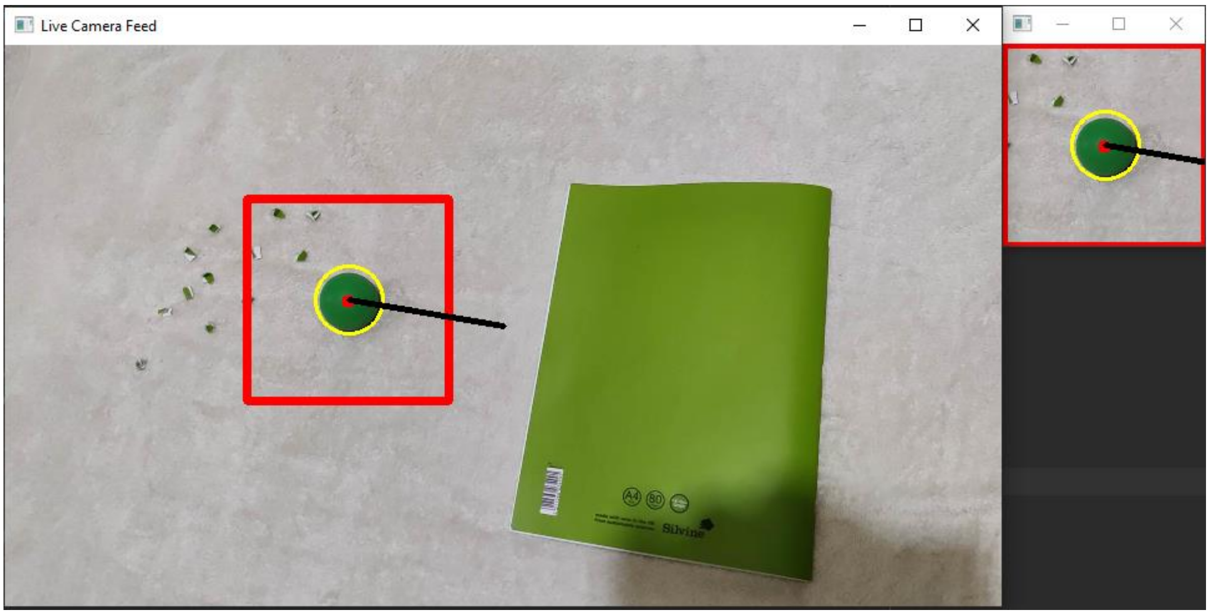


Figure 3-41 - Example with secondary window to show the working area.

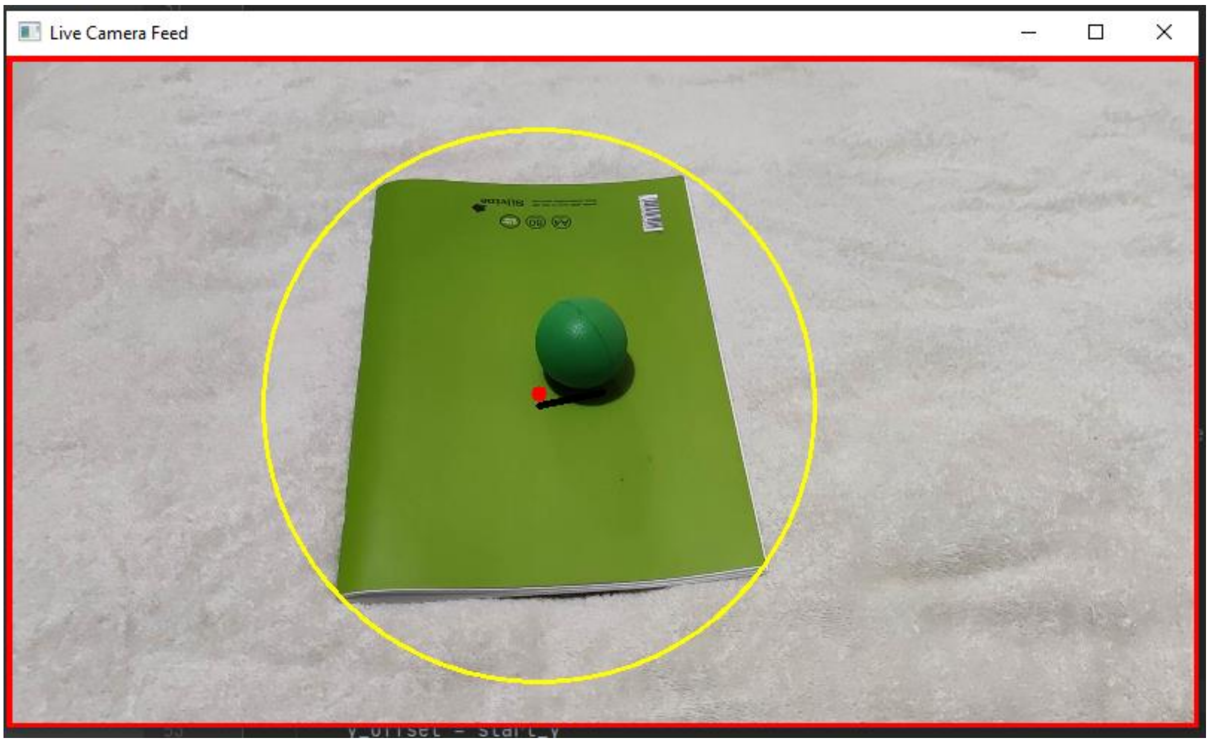


Figure 3-42 - Occlusion issue.

While this local area frame helps with keeping lock on the currently tracked object, it does nothing for occlusion with other larger background objects.

Section 3.8 discusses future work improvements that could assist with this type of occlusion handling.

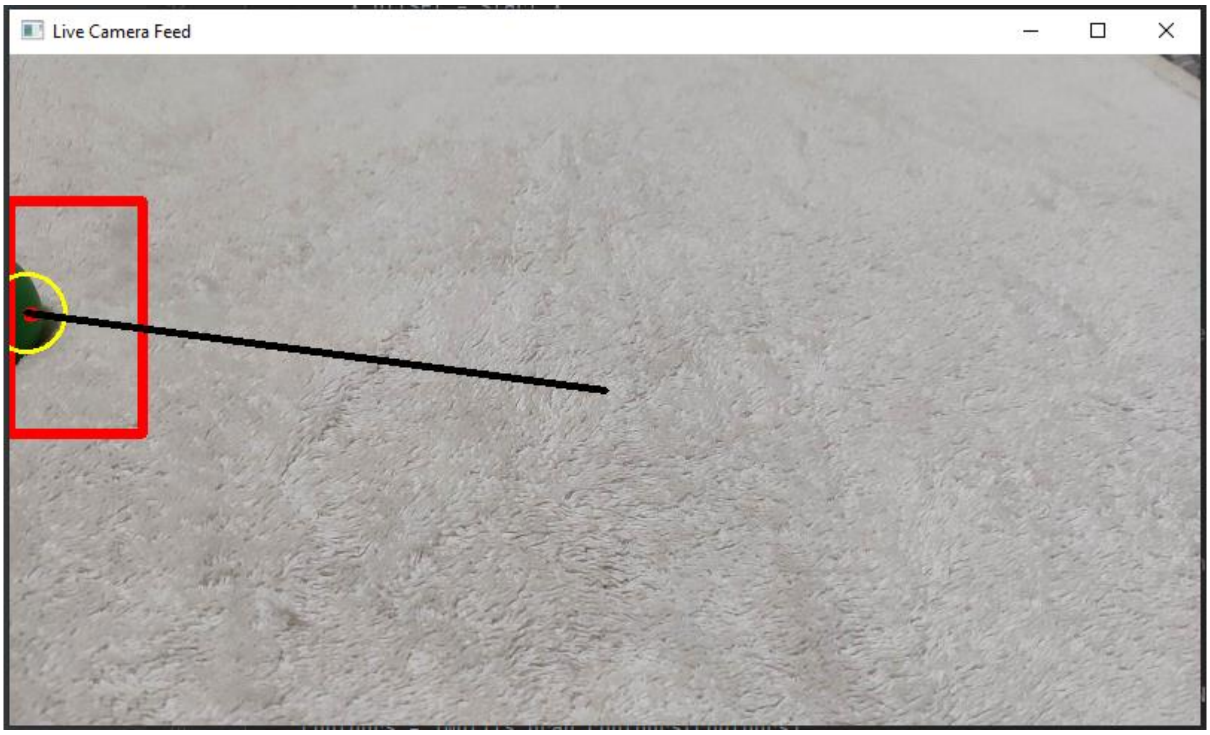


Figure 3-43 - Edge sizing issue.

As the object approaches the edge of the frame, the contour it creates in the binary mask is significantly smaller due to it simply not being in frame, this causes the program to assume the object is further than it is meant to be, thus lowering the volume significantly as it approaches the edge.
While this is a minor issue, it is not much of a problem as the audio is meant to inherit the line-of-sight requirements from vision, so all this issue amounts to is a fade out of volume rather than an instant cut out.

3.8 Future work and improvements

Edge detection would allow the tracker to work with textures alongside colour, improving accuracy significantly and removing most possibilities for background objects to occlude the target object. This would also open opportunities to track multiple objects without major occlusion issues.
The main downside of using edge detection with colour detection would be the computational power required to run it in real time, a way around this could be to have it toggled.
If the program detects background occlusion on the object being tracked, it could switch to edge detection temporarily to keep the object tracked.
Detecting occlusion can be as easy as checking for sudden jumps in object size or position.

Edge detection also gets around the issue of the tracking currently being limited to objects with a consistent cross-section for accurate tracking from any visual angle, as distance calculation is purely based on 2-dimensional width, of which most objects do not have a consistent width when you rotate them:

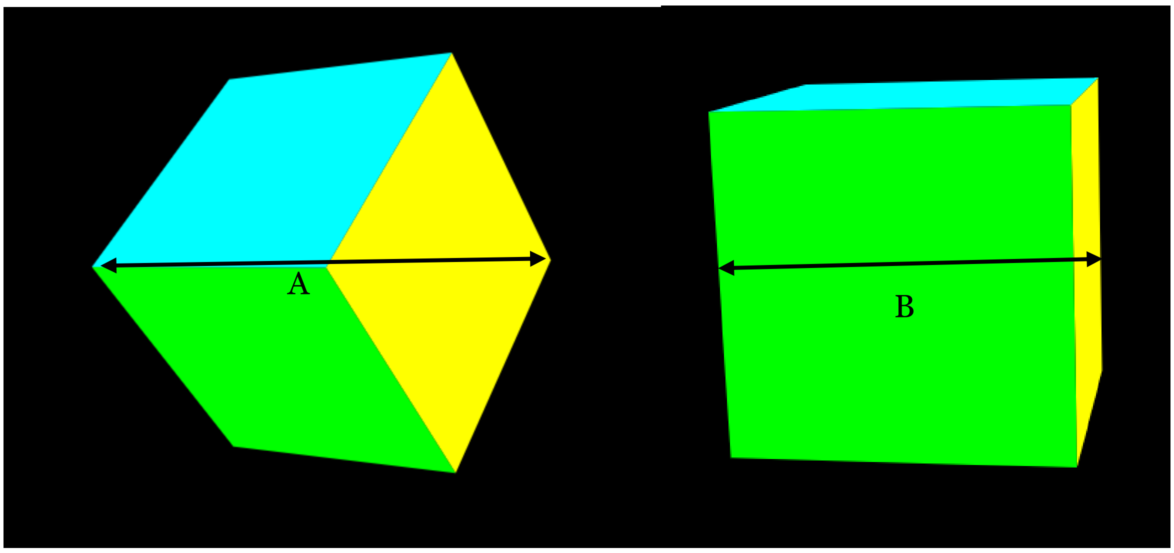


Figure 3-44 - Non-sphere objects being rotated. (A>B)

Velocity calculations on the tracked object could also prove useful for defining the local area frame dimensions. For an object not moving fast, the area could be significantly reduced which therefore reduces the processing area and the area where a larger object could interfere. The area could get larger as the object moves faster to ensure the distance travelled per frame does not put the currently tracked object outside the boundaries of the local area.
Once the object has gone off-screen, rather than snapping the local area frame back to the default frame (allowing the program to snap onto another target), the local area frame could grow at a steady rate from its current position until it reaches the default frame size. This would allow an object to hop out of frame and make its way back in roughly the same location without instantly switching away from it.

These velocity calculations can also be used to predict movement when being foreground occluded by another object.
To further emphasise the object's velocity, audio pitch can be increased/decreased to match the object's velocity relative to the listener.

3-4 Cameras or a single 360-degree camera could be used instead to have full coverage around the listener, although the line-of-sight limited audio has possible uses such as assisting visually impaired individuals get around busy cities similar to how car parking proximity detectors would beep when getting too close to obstacles, except it could be context based and grant the listener more tailored information based on surroundings.

4. Testing and Summary

4.1 Introduction

Three accuracy tests were conducted twice, once with artificial noise and one without:

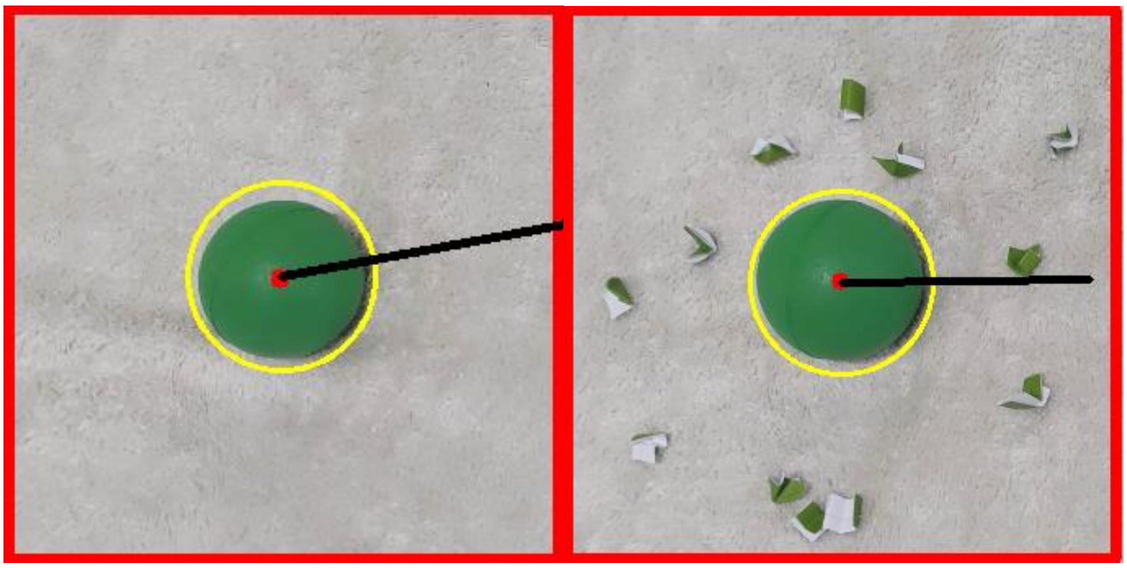


Figure 4-1 - Noiseless and artificial noise test conditions.

These tests were conducted using my personal mobile device's ultra-wide camera with a viewing angle of 123°, a cheap laser distance measuring device and a tripod. The first test was a simple distance calculation test similar to the experiment in section 2.9.

The second and third test had the camera move horizontally right to left, causing the object onscreen to appear as moving left to right, the horizontal distance from the centre position was measured both in real-world and calculated in the program. The second test started at 50cm distance from the object, the third test at 100cm.

The second and third tests assume that at laser distance zero, the program is also at zero distance because the program calculates using this exact assumption, any value obtained from measuring at zero would be purely human error.

4.2 First test data

(cm) With artificial noise				(cm) Noiseless			
Laser	Program	Difference		Laser	Program	Difference	
10	9.901	-0.099	0.9999%	10	9.943	-0.057	0.5733%
20	20.22	0.220	1.0880%	20	20.138	0.138	0.6853%
30	30.121	0.121	0.4017%	30	30.154	0.154	0.5107%
40	39.984	-0.016	0.0400%	40	40.167	0.167	0.4158%
50	50.096	0.096	0.1916%	50	50.122	0.122	0.2434%
60	60.084	0.084	0.1398%	60	59.987	-0.013	0.0217%
70	70.103	0.103	0.1469%	70	69.954	-0.046	0.0658%
80	80.139	0.139	0.1734%	80	80.07	0.070	0.0874%
90	89.895	-0.105	0.1168%	90	90.213	0.213	0.2361%
100	100	0.000	0.0000%	100	100	0.000	0.0000%

Table 4-1 - Central distance measurements with and without artificial noise.

In the following graphs, the blue lines refer to the data with artificial noise and the orange lines refer to the noiseless data.

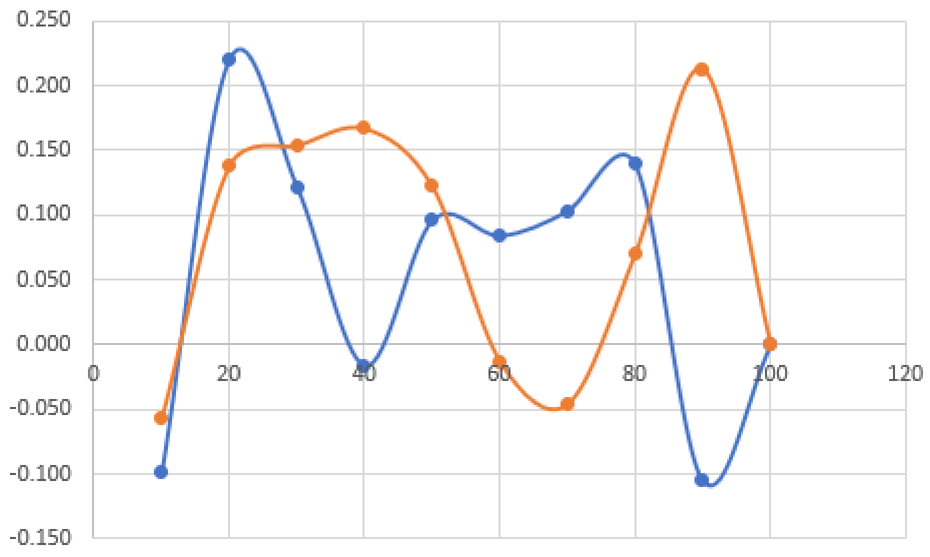


Figure 4-2 - Graph of differences between measured central distances (cm) and program calculated central distances (cm).

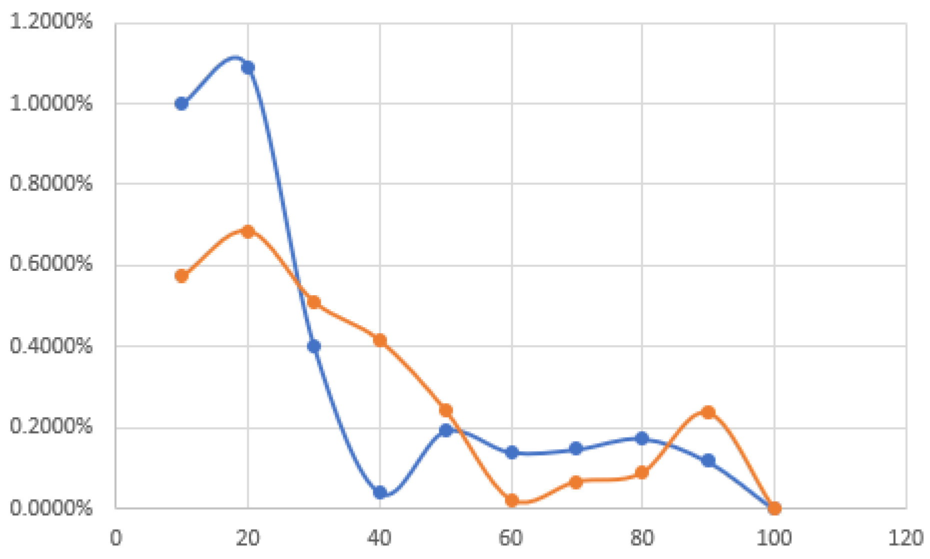


Figure 4-3 - Graph of percentage differences between measured central distances (cm) and program calculated central distances (cm).

4.3 Second test data

(cm) With artificial noise				(cm) Noiseless			
Laser	Program	Difference		Laser	Program	Difference	
-25	-26.31	-1.310	4.9791%	-25	-26.41	-1.410	5.3389%
-20	-21.116	-1.116	5.2851%	-20	-21.66	-1.660	7.6639%
-15	-15.849	-0.849	5.3568%	-15	-16.57	-1.570	9.4750%
-10	-10.482	-0.482	4.5984%	-10	-11.13	-1.130	10.1527%
-5	-5.11	-0.110	2.1526%	-5	-5.73	-0.730	12.7400%
0	0	0	0	0	0	0	0
5	5.24	0.240	4.5802%	5	5.11	0.110	2.1526%
10	10.63	0.630	5.9266%	10	10.53	0.530	5.0332%
15	16.03	1.030	6.4255%	15	15.97	0.970	6.0739%
20	21.28	1.280	6.0150%	20	21.33	1.330	6.2353%
25	26.75	1.750	6.5421%	25	26.81	1.810	6.7512%

Table 4-2 - Horizontal distance measurements at 50cm central distance with and without artificial noise.

In the following graphs, the blue lines refer to the data with artificial noise and the orange lines refer to the noiseless data.

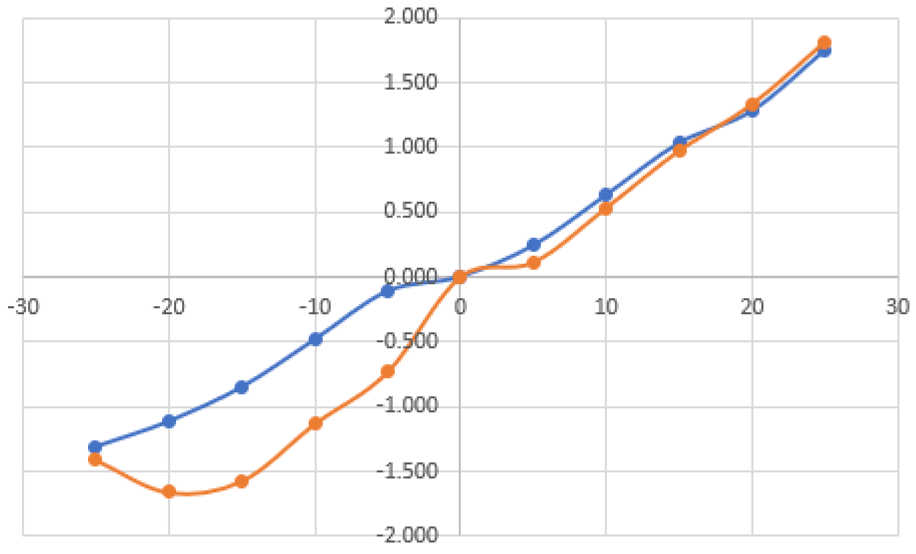


Figure 4-4 - Graph of differences between measured horizontal distances (cm) and program calculated horizontal distances (cm) at 50cm central distance.

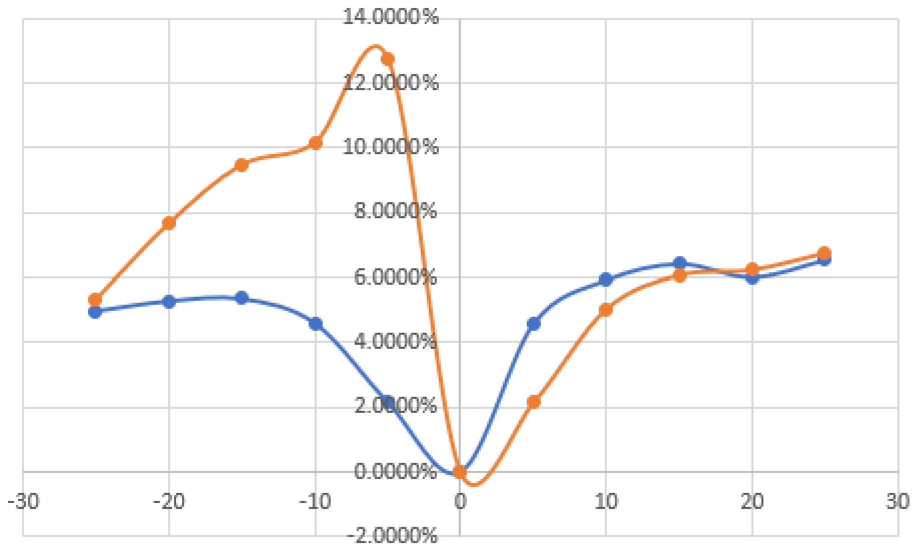


Figure 4-5 - Graph of percentage differences between measured horizontal distances (cm) and program calculated horizontal distances (cm) at 50cm central distance.

4.4 Third test data

(cm) With artificial noise				(cm) Noiseless			
Laser	Program	Difference		Laser	Program	Difference	
-50	-56.95	-6.950	12.2037%	-50	-55.43	-5.430	9.7961%
-40	-44.61	-4.610	10.3340%	-40	-44.49	-4.490	10.0922%
-30	-33.27	-3.270	9.8287%	-30	-33.35	-3.350	10.0450%
-20	-22.93	-2.930	12.7780%	-20	-22.54	-2.540	11.2689%
-10	-11.59	-1.590	13.7187%	-10	-10.67	-0.670	6.2793%
0	0	0	0	0	0	0	0
10	10.09	0.090	0.8920%	10	10.21	0.210	2.0568%
20	20.43	0.430	2.1047%	20	21.15	1.150	5.4374%
30	31.77	1.770	5.5713%	30	32.12	2.120	6.6002%
40	43.11	3.110	7.2141%	40	43.43	3.430	7.8978%
50	54.45	4.450	8.1726%	50	53.86	3.860	7.1667%

Table 4-3 - Horizontal distance measurements at 100cm central distance with and without artificial noise.

In the following graphs, the blue lines refer to the data with artificial noise and the orange lines refer to the noiseless data.

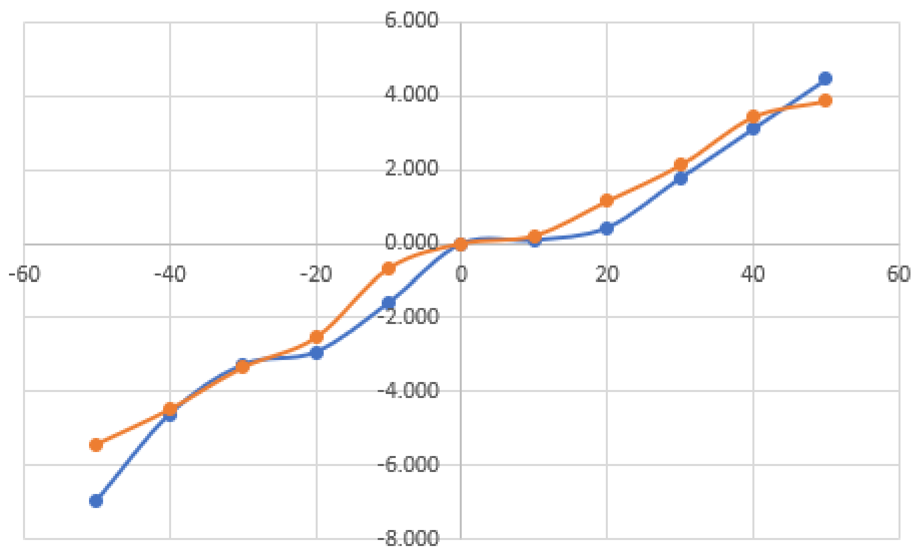


Figure 4-6 - Graph of differences between measured horizontal distances (cm) and program calculated horizontal distances (cm) at 100cm central distance.

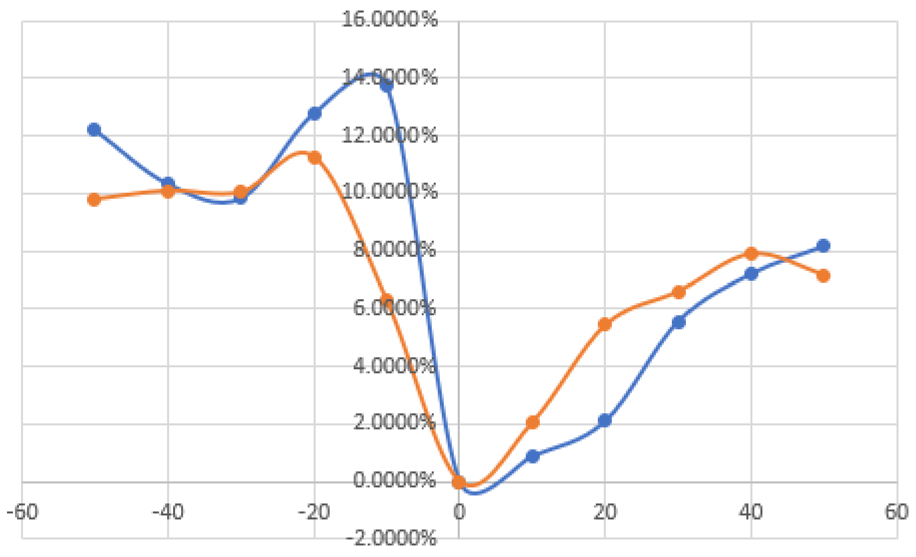


Figure 4-7 - Graph of percentage differences between measured horizontal distances (cm) and program calculated horizontal distances (cm) at 100cm central distance.

4.5 Summary

The data collected shows central distance calculations to be almost an order of magnitude more accurate than any horizontal distance calculations. This was expected as the horizontal distance calculations are done using the data from the central distance calculations and thus inherits all potential error in that value. The absolute error in horizontal distance scales with how far the object is from the camera with a 6.95cm error at 100cm central distance compared to a maximum error of 1.81cm at 50cm central distance.

The main shortcomings of the program being that of a limit set of trackable objects and occlusion issues are both opposed by its reliable and incredibly fast tracking.

References

[1] J. F. Henriques, R. Caseiro, P. Martins and J. Batista, "High-Speed Tracking with Kernelized Correlation Filters," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 37, no. 3, pp. 583-596, 1 March 2015, doi: 10.1109/TPAMI.2014.2345390.

[2] Z. Kalal, K. Mikolajczyk and J. Matas, "Tracking-Learning-Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, no. 7, pp. 1409-1422, July 2012, doi: 10.1109/TPAMI.2011.239.

[3] Z. Kalal, K. Mikolajczyk and J. Matas, "Forward-Backward Error: Automatic Detection of Tracking Failures," 2010 20th International Conference on Pattern Recognition, Istanbul, 2010, pp. 2756-2759, doi: 10.1109/ICPR.2010.675.

[4] D. S. Bolme, J. R. Beveridge, B. A. Draper and Y. M. Lui, "Visual object tracking using adaptive correlation filters," 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Francisco, CA, 2010, pp. 2544-2550, doi: 10.1109/CVPR.2010.5539960.

[5] B. Babenko, M. Yang and S. Belongie, "Visual tracking with online Multiple Instance Learning," 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, 2009, pp. 983-990, doi: 10.1109/CVPR.2009.5206737.

[6] Held, David, Sebastian Thrun, and Silvio Savarese. "Learning to track at 100 fps with deep regression networks." In European Conference on Computer Vision, pp. 749-765. Springer, Cham, 2016.

[7] Lukezic, Alan, Tomas Vojir, Luka ˇCehovin Zajc, Jiri Matas, and Matej Kristan. "Discriminative correlation filter with channel and spatial reliability." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 6309-6318. 2017.

[8] pyimagesearch "Triangle Similarity for Object/Marker to Camera Distance" [Online]. Available: https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/

[9] Open Source Computer Vision Documentation “Basic Thresholding Operations” [Online]. Available: https://docs.opencv.org/3.4/db/d8e/tutorial_threshold.html

[10] IP Webcam Mobile Application [Online]. Available: https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en&gl=US

[11] OpenCV-Python "OpenCV-Python Documentation" [Online]. Available: https://docs.opencv.org/master/d6/d00/tutorial_py_root.html

[12] Imutils “Imutils Home Page” [Online]. Available: https://pypi.org/project/imutils/

[13] Numpy “Numpy Home Page” [Online]. Available: https://numpy.org/

[14] Pyglet “Pyglet Home Page” [Online]. Available: http://pyglet.org/

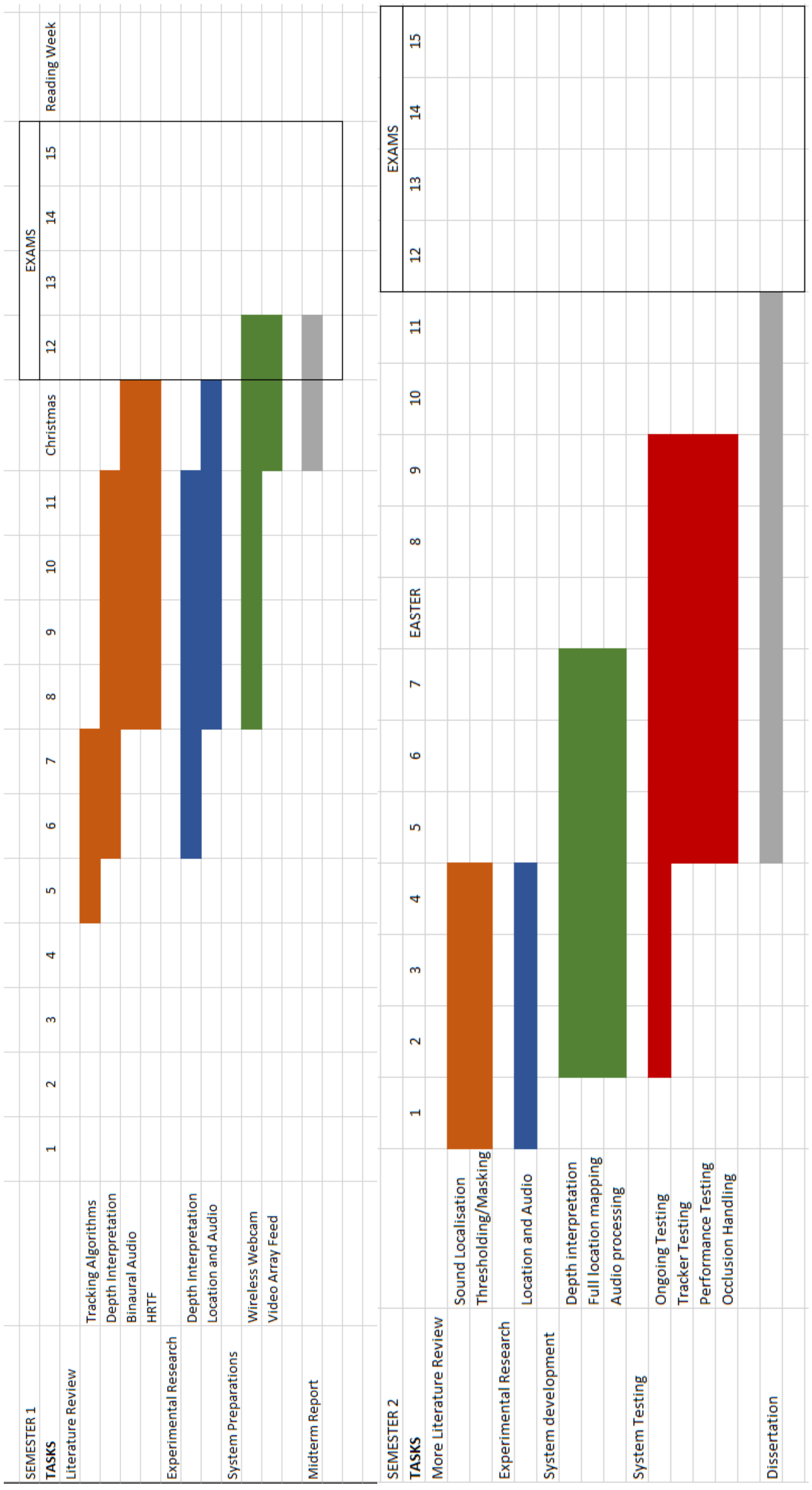
[15] OpenAL “OpenAL Home Page” [Online]. Available: https://openal.org/

[16] Ffmpeg “ffmpeg Home Page” [Online]. Available: https://ffmpeg.org/

[17] Python 3.9 “Python Home Page” [Online]. Available: https://www.python.org/

[18] PyCharm Community Edition “PyCharm Home Page” [Online]. Available: https://www.jetbrains.com/pycharm/

Appendix


SEMESTER 1
TASKS
Literature Review
Tracking Algorithms
Depth Interpretation
Binaural Audio
HRTF
Experimental Research
Depth Interpretation
Location and Audio
System Preparations
Wireless Webcam
Video Array Feed
Midterm Report
1
2
3
4
5
6
7
8
9
10
11
Christmas
12
13
14
15
EXAMS
Reading Week
SEMESTER 2
TASKS
More Literature Review
Sound Localisation
Thresholding/Masking
Experimental Research
Location and Audio
System development
Depth interpretation
Full location mapping
Audio processing
System Testing
Ongoing Testing
Tracker Testing
Performance Testing
Occlusion Handling
Dissertation
1
2
3
4
5
6
7
EASTER
8
9
10
11
12
13
14
15
EXAMS