Interactive Projections Using Computer Vision

School of Information Technology and Electrical Engineering (ITEE)

Author
E.S. Alexander

Supervisor
Dr Surya P. N. Singh

November 4, 2019

E.S. Alexander
sandman.esalexander@gmail.com

November 4, 2019

Prof. Amin Abbosh
Head of School (Acting)
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Qld 4072

Dear Professor Abbosh,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Mechatronic Engineering, I present the following thesis entitled

"Interactive Projections Using Computer Vision"

This work was performed under the supervision of Dr Surya Singh.
I declare that the work submitted in this thesis is my own, except as noted within, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,
E.S. Alexander

Acknowledgements

This thesis would not have been possible without the guidance, expertise, and enthusiasm of my supervisor, Dr Surya P.N. Singh. Your course-corrections and realistic expectations helped keep the project on target, instead of an arrow aimed too high, crashing back down on the archer. I highly appreciate the opportunity to have worked on a project I was truly interested in - thanks for your patience as I bounced between ideas of all flavours.

Thank you to my faithful testers, for your patience and multiple test runs as bugs and system modifications were addressed. Your feedback helped focus the project on the most important areas, and your friendship, discussion, encouragement, and support throughout the project has been invaluable.

Thank you also to my family, for your continual support throughout the year (and all the years leading up to it), and for helping to proofread this document. Your attention to detail and focus on quality and understanding has helped immeasurably throughout my education, and drives me to do good work for my own sake rather than just to meet a metric.

Thank you finally to my partner, for sharing my excitement, and for bringing it back in moments of doubt. Your belief in me and this project kept it afloat despite the rocks it scraped along the way.

Abstract

To help facilitate dynamic and interactive presentations, and interactive projections, an open-source framework for detection and management of projector screen interactions is proposed, using a webcam for feedback. The framework is modular and expandable, and includes a proof-of-concept implementation of automatic screen detection, laser-pointer detection, Kalman compensation, and the use of the `pyautogui` Python library to handle control of a computer mouse as a result of detected interactions.

The target implementation is shown to be feasible for computers up to 2.67 times slower than the 15" mid-2015 Macbook Pro test device, robustly detecting projector screens and tracking laser-pointer interactions. The base-implementation runs at approximately 30 FPS on the test device, and the system was still usable at down to 12.5 FPS.

Projector screen detection exploits temporal control over the projection screen and camera to easily detect a screen within the camera image in a static environment. The screen corners are detected as intersections of the detected edges of the screen. From these, a projective transform is generated mapping the screen region of a camera image to the optionally downsampled screen resolution.

Interactions are detected as distortions on the expected background image, using a combination of reference lighting and the known screen image being displayed. Distortions are characterised using both colour and shape information for robustness.

To better account for the true position of an interaction, an extended Kalman filter is used for compensation, assuming a constant velocity from the last measurement. This helps to reduce the detrimental effects of system delay, and also allows for continued interaction motion through short periods of missed detection. Compensation was shown to improve high-fidelity performance at maximum speed by up to 20%.

This framework enables dynamic and interactive presentations, while also providing a basis for further projector interaction methods and detection techniques. With refinement, the framework could allow for low-cost, scalable, and transportable projection interaction technology, for use in streamed and recorded interactive presentations, artistic and digital design work on large interactive screens, and as an accessible alternative to existing costly smart-board technology.

The project is maintained at:
https://github.com/ES-Alexander/interactive-projectivity-open

Contents

1 Introduction **1**
- 1.1 Motivation for the Project . . . 1
- 1.2 Aim and Scope of the Project . . . 2
- 1.3 Structure of the Thesis . . . 2

2 Literature Review **3**
- 2.1 Screen Detection . . . 3
- 2.2 Laser Pointer Detection . . . 4
- 2.3 Existing Open-Source Functionality . . . 4
  - 2.3.1 Hardware Connection . . . 4
  - 2.3.2 Computer Vision . . . 4
  - 2.3.3 Screenshots . . . 4
  - 2.3.4 Device control . . . 5

3 System Design and Methodology **6**
- 3.1 System Components . . . 6
  - 3.1.1 Device Properties . . . 7
- 3.2 Program States . . . 8
- 3.3 Screen Detection and Calibration . . . 9
- 3.4 Interaction Detection . . . 12
- 3.5 Compensation . . . 14
  - 3.5.1 Utility of Compensation . . . 14
  - 3.5.2 Compensator Selection . . . 15
  - 3.5.3 Compensator Implementation and Tuning . . . 16
- 3.6 Further Considerations . . . 17

4 Results/Analysis **18**
- 4.1 Sufficiency . . . 19
- 4.2 Baseline Comparison . . . 20

5 Conclusion and Recommendations **22**

A Controller Code **25**

B Calibration and Testing Code **33**

C Screen Detection Code **38**

D Edge Processing Code **45**

E Interaction Detection Code **63**

F Data Analysis Code **69**

List of Figures

3.1 System Components and Chosen Implementation 7
3.2 Algorithm Process Flow 8
3.3 Screen Calibration Steps (pt 1) 10
3.4 Screen Calibration Steps (pt 2) 11
3.5 Laser Detection Steps (pt 1) 13
3.6 Laser Detection Steps (pt 2) 14

4.1 Maze Sample Trial 18
4.2 Screen Detection Example 19
4.3 Equivalent Frames Per Second Processing Speed 19
4.4 Effect of Added Delay on Usability (Fidelity at Speed) 20
4.5 Compensated Tracking Maze Completion Time Benefit 21

List of Tables

2.1 Screenshot Library 5
2.2 Device Control Library 5

3.1 Tuned noise variance 16

Chapter 1

Introduction

1.1 Motivation for the Project

In our modern, technological society, we strive for our machines and devices to meld seamlessly with the world around us. These augmentations of reality work to bring us enhanced lived experience, and enable us to interact with our environment, where even a static, plain surface can seem to be dynamic and easily influenced by our actions.

With projection and screen systems ubiquitous, significant research has gone into making our displays interactive. External peripherals such as keyboards, mouses, buttons, and joysticks allow for effective interaction at a low cost, but constrain the operator to the device. Touch-screen technologies add freedom of motion, but scale badly because they require significant increases in sensor numbers for larger screens, and are also generally limited to installed screens that cannot be easily moved or transported. This is true of common resistive and capacitive in-screen technologies, as well as various infrared (IR) light emitter-detector systems, such as the in-screen Distributed Infrared (DIR), Digital Vision Touch (DViT), and Hybrid Precision (HyPr) Touch by SMART technologies [1, 2, 3]. There are also remote sensing touch-screen systems, which are often scalable but require costly equipment which is generally installed in a single place. Examples of these systems are the use of multiple depth cameras in [4], or an external IR emitter with a linked IR camera detecting shadows and reflections cast by the operator, and registering touches on detected tip collisions of the shadow and reflection [5].

To overcome the limitations of scalability, cost, and transportability requires a remote, low-cost, movable sensor, subject more to algorithmic complexity and processing capabilities than display size and sensor numbers. To meet these objectives this thesis focuses on using computer vision with a single standard camera to detect and register user interactions with a projector screen. If implemented to an advanced capacity, this could form a low-cost alternative to existing smartboard technologies, enable remote interaction in presentations without constraining the presenter to a computer, and provide artists and designers with a transportable, large interactive workspace in the form of a laptop and portable projector.

As a proof of concept of the technology, this thesis focuses on presentations, whereby a presenter can shine a laser-pointer to remotely interact with their display, while simultaneously giving a dynamic presentation. This helps solve the problem that when presenting

material on multiple screens, presenters must currently either constrain themselves to their presentation device, or present dynamically for only those audience members in the room. Even in large presentation venues such as lecture theatres with two screens, using a laser-pointer ordinarily means only one screen displays what the presenter is referring to. The system proposed here allows the detection and display of a laser pointer on all screens, including for those watching a live-stream, or a recording afterwards.

Thus, this project focuses on addressing the question:
Is it possible for a presenter to remotely interact with their presentation in a dynamic manner, such that all audience members, present or streaming, have a similar experience?

1.2 Aim and Scope of the Project

The aim of this project is to create an open-source framework for detecting and managing presenter interactions with a projector screen.

The implementation scope is presentation environments, with a projector screen detected using a webcam feed. This allows proof of concept using a cheap, ubiquitous remote sensor that is already available on most presentation computers. The target implementation restricts interactions to a laser-pointer, which is detected from the stream in real-time using computer vision. Laser-pointer interactions are expected to be relatively easy to detect, allowing development and demonstration of the framework within a limited time frame, while also providing a sense of the computation speed requirements for more advanced implementations. Detection of multiple interactions simultaneously, and the translating of interactions into gestures are beyond the scope of this thesis.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows:

Chapter 2. Literature Review: Consideration of similar systems and technologies, and existing available code.

Chapter 3. System Design and Methodology: A description of the required and useful system components, as well as the selected implementation functionality and details, including an exploration of compensation to improve the performance of the system in adverse detection conditions.

Chapter 4. Results/Analysis: The results from testing the system, from the experience of an expert and two novices, including a comparison of the compensated system to its uncompensated baseline.

Chapter 5. Conclusions and Recommendations A summary of the state of the project with regard to its requirements, and recommendations for further advancement of the system.

Chapter 2

Literature Review

In this chapter, relevant literature is considered that discusses systems for interaction with a projector screen, with similar requirements and behaviour as required by the project aim and scope. Accordingly, this chapter identifies the abstract requirements of the system, and existing literature on similar topics.

2.1 Screen Detection

Screen detection is an essential part of a computer-vision interaction framework, because it is required for determining where interactions are within the screen. The simplest method of determining this is allowing the user to specify the screen corners on a displayed camera image. This, however, is error-prone, unreliable, and inconvenient, so an automated detection system is preferred.

One option proposed in [6] suggests a scenario whereby "the system uses the local feature method SURF (speeded-up robust features) to find some interest points before applying the RANSAC paradigm (Random Sample Consensus) to automatically generate the homography matrix [between the camera and screen]." This is effective, but is somewhat complex and is reliant on evaluating a probability criterion until within a tolerance.

An alternative option consists of drawing horizontal and vertical lines on a projector screen one at a time, and taking photos of them as part of a keystone correction. The camera-screen homography is then determined by considering the intersection points between pairs of horizontal and vertical lines. [7]

From this, it is understood that screen detection can be accomplished most easily by exploiting control of the projector over time, and also using a known image to determine location. Once the screen is detected, a homography can be made mapping the camera image to the screen, which allows for a projective transformation from the image to just the screen rectangle.

2.2 Laser Pointer Detection

Once the screen has been detected, the laser pointer detection and tracking can begin, using a variety of potential methods. A trivial solution is searching for the brightest correct-coloured spot on the screen, as suggested in [8], although this can be problematic with the laser saturating the camera and appearing as white. An alternative is presented in [7], whereby an initial image is taken with the projection displayed as white, and a threshold is used to determine spots on the image that are sufficiently brighter than the maximal brightness background from the white screen image. This has issues if lighting changes significantly while running the tracking, but is otherwise quite robust to dynamic backgrounds. Both these methods make use of a multi-stage search, where a small region around the last detected position of the laser is searched first, and if the laser is not detected within that region the rest of the image is searched with a convolution.

A more recent detection method is described in [6], where camera images are converted to hue-saturation-intensity colour-space to allow for more accurate colour-matching of the laser while also thresholding on intensity. This is less affected by lighting changes where the laser is bright enough to maintain its colour, and is a relatively robust detection method.

From these methods it can be understood that the primary features for extracting laser pointer position from an image are intensity, colour, shape, and exploitation of known properties of the projected screen.

2.3 Existing Open-Source Functionality

Some desired functionality of the code is already freely available as part of other open-source projects. To maximise on this, Python was selected as the implementation programming language due to its cross-platform compatibility, ease of use, and extensive availability of libraries.

2.3.1 Hardware Connection

In order to connect the control program to a webcam feed, `openCV` was installed. As a cross-platform open-source programming library for computer vision, `openCV` allows connecting to most webcams and many other cameras.

2.3.2 Computer Vision

To perform any kind of image processing and object detection requires computer vision functionality. This is provided using `openCV`, with `numpy` used to perform fast array operations.

2.3.3 Screenshots

To best exploit known information about the projected screen, it is helpful to be able to capture the current screen with a screenshot. Table 2.1 displays the two main options for this, with `mss` proving superior in both platform support and speed.

Table 2.1: Screenshot Library

Library	**platforms?**	**notes**
`Pillow`	no linux	slow
`mss`	all	fast

2.3.4 Device control

Interactions are required to not only be detected, but should also have some influence over the source. To do this requires device control, which is most easily achieved by emulating a mouse and/or keyboard. The primary purpose for the target implementation is mouse control, but in the interests of allowing more complex interactions in future it is preferable to allow more interaction methods where possible. Table 2.2 compares the major options for device control, with `pyautogui` selected over `Pynput` due to less overhead from not implementing listening behaviour, and selected over `autopy` because `pyautogui` is pure Python and easier to install.

Table 2.2: Device Control Library

Library	**platforms?**	**mouse**	**keyboard**	**notes**
`Pynput`	all	1	1	control+listen
`pyautogui`	all	1	1	control
`mouse`	no macOS	1	0	
`autopy`	all	1	1	rust-based
`PyUserInput`	all	1	weak	

Chapter 3

System Design and Methodology

This chapter provides the details and justification for the target implementation, used to create the desired interaction detection and managing framework. It discusses the components of the system, flow of execution through the program states of the controller, and further considerations that could improve the system in future.

3.1 System Components

The proposed system is comprised of the following components, which are required for an interaction controller to function. The connection of components and their translation to the physical implementation are displayed in Figure 3.1.

- **Sensor:** Collects information about the environment, including the display signal being interacted with
- **Interaction Detector:** Extraction of interaction locations and timing
- **Controller:** Translation of interactions into influence on the source
- **Source:** The signal being displayed
- **Display:** The physical presentation of the source within the display environment, to be interacted with
- **Interaction Distortions:** Intentional distortions of the display signal recorded by the sensor, used to interact with the display

The sensor itself will also bring noise into the system, which is exacerbated by any misalignment of the detected screen region.
From the base components, the figure outlines the target implementation, which uses computer vision on a laptop to detect laser-pointer distortions on a projector screen, as viewed by the laptop webcam.

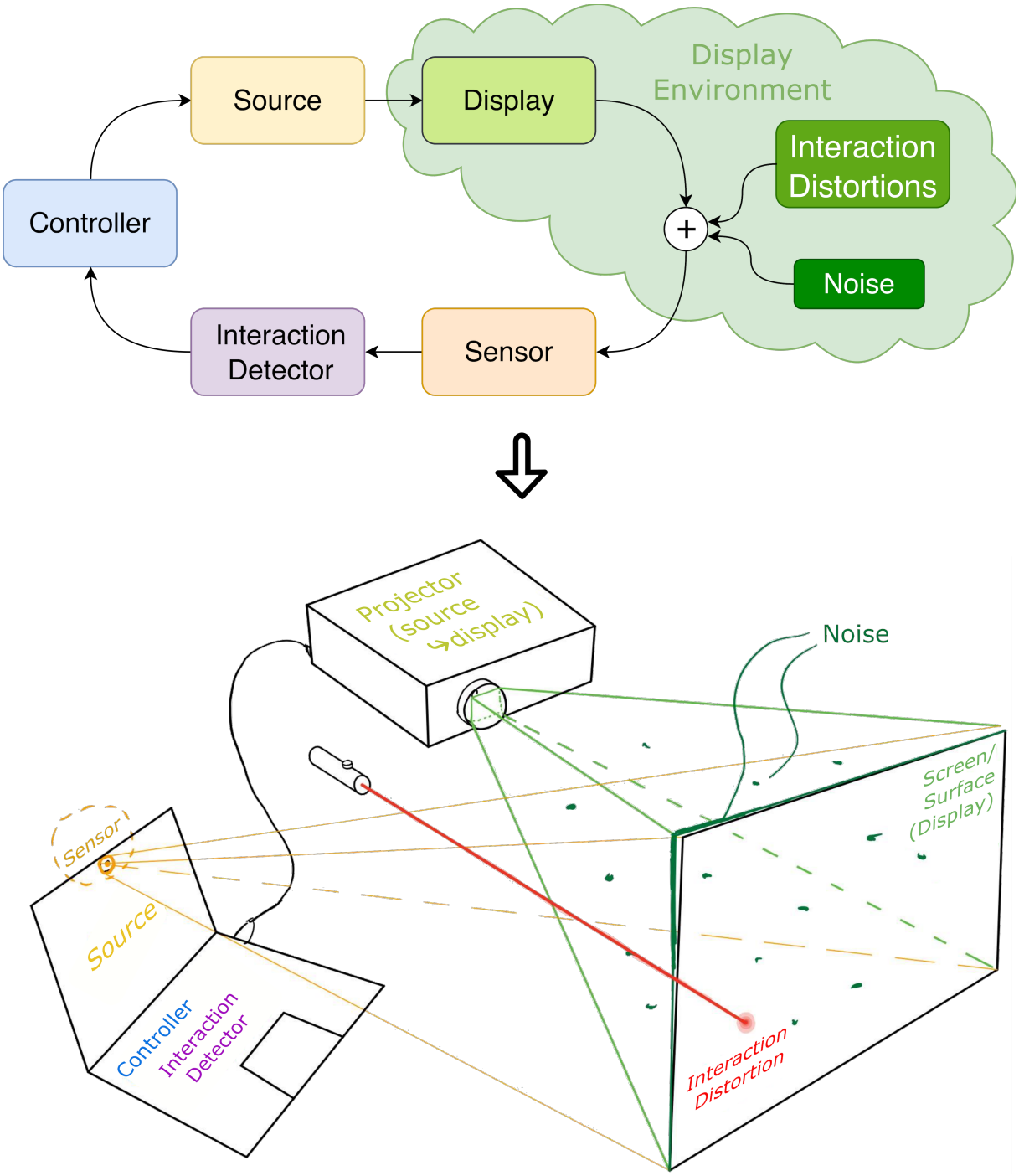


Figure 3.1: System Components and Chosen Implementation

3.1.1 Device Properties

All implementation and testing was performed using a 15" mid-2015 Macbook Pro, with a 2.5GHz Quad-Core Intel core i7 processor, Intel Iris Pro 1536 MB graphics augmented with an AMD Radeon R9 M370X 2 GB, and 16 GB 1600 MHz DDR3 RAM.

3.2 Program States

From the main components of the system, it is possible to split the detection and control algorithm into related program states, as provided in Figure 3.2. These include a breakdown into required components of each state, helpful optional components which are implemented, and components which could be helpful but have not been implemented in the current codebase (shown in red). State colouring is provided linking the program states to the respective system components from Figure 3.1. The main control algorithm code can be found in Appendix A.

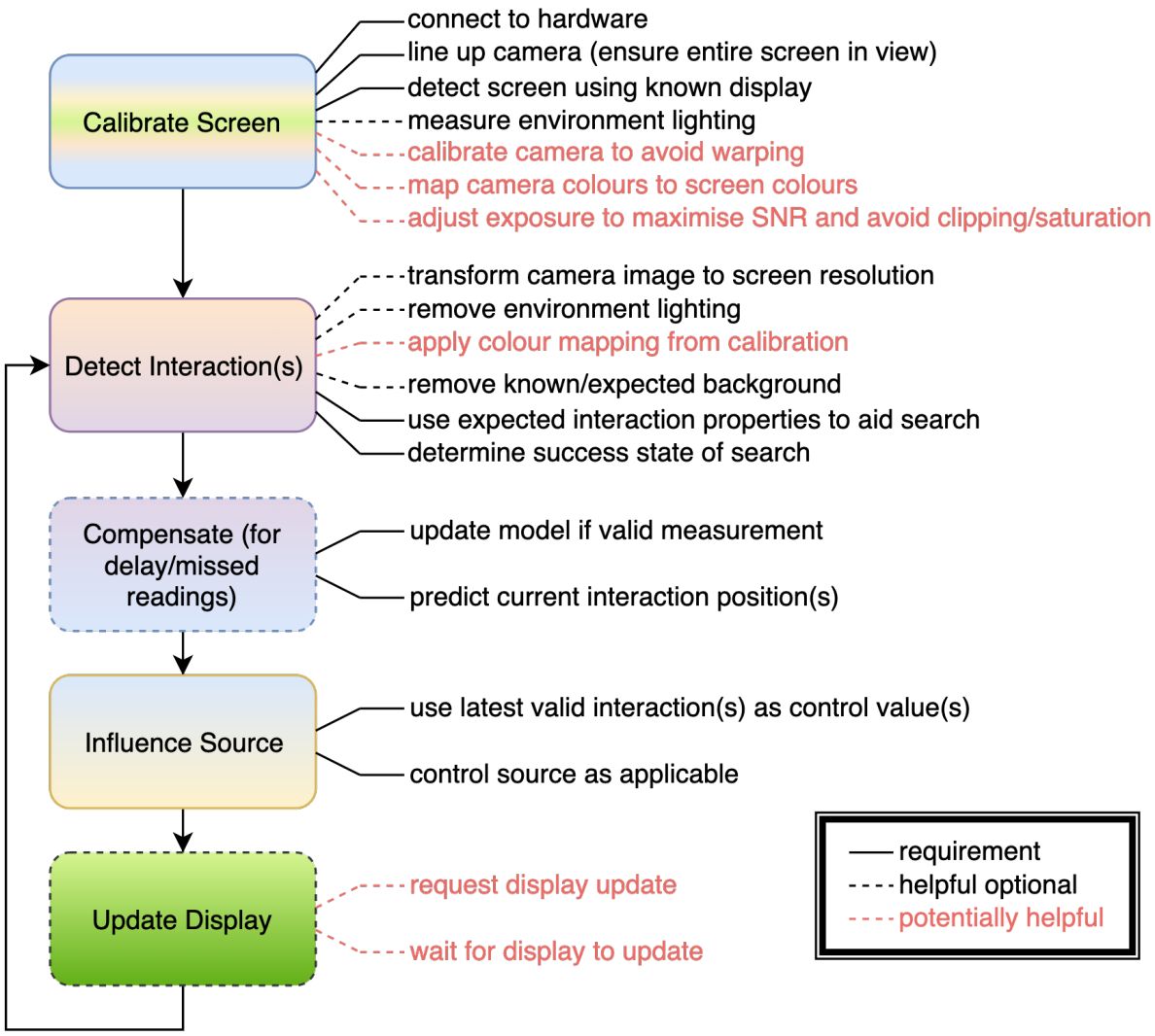


Figure 3.2: Algorithm Process Flow

3.3 Screen Detection and Calibration

As presented in the states above and in the literature exploration, screen detection must occur before the main control loop and the interaction detection can begin. Finding the screen in the simplest case requires finding the four corner points, from which a projective transform can be made mapping the screen region of future images from the camera into just the screen region, for direct comparison with screenshots of the laptop screen. For this to be successful requires relatively limited lens warping from the camera, which is a reasonably valid assumption now as compared to when the literature resources were published. This allows some simplification of the detection process in that only a simple transform is required to be developed.

The following list outlines the steps implemented to robustly detect the screen, making use of control over the display through time to remove the background and make the detection as easy as possible. The code for screen calibration and detection is found in Appendices B and C, with edge processing code in Appendix D.

1. **Setup:** User connects hardware and ensures screen is fully within camera view.
2. **Capture Background:** Set the screen to black and take an image (effectively without the screen).
3. **Capture Screen:** Set the screen to white and take an image.
4. **Remove Background:** Subtract the known background from the screen image to leave just the screen and noise. Negative values in the result are set to 0, as a camera cannot capture negative light.
5. **Reduce Noise:** Slightly blur the high-contrast screen image by convolving with a Gaussian kernel, to reduce spikes from noise.
6. **Greyscale:** Convert the image to a greyscale intensity map to prepare for binarisation.
7. **Binarise:** Convert to a binary image using Otsu's method, assuming a bi-modal intensity distribution (screen/no screen) and thresholding between the modes.
8. **Edges:** Use Canny edge-detection to extract all edges from the binary image.
9. **Lines:** Find straight lines in the edge image using the Probabilistic Hough Transform, then scale detected lines to the image boundaries - these should be just the edges of the screen.
10. **Line Reduction:** Combine lines that intersect within the image with an angle of less than 5 degrees between them - these are assumed to be along the same screen edge.
11. **Intersections:** Get the intersection points between all the lines.
12. **Screen Corners:** The screen region is defined by 4 corner points - if more than 4 points are available, use k-means clustering to determine the most likely corner locations. Less than 4 detected points registers as a failed calibration.

13. **Ordering:** Assuming the camera and screen have a similar or identical vertical axis direction, order the corner points consistently according to their position relative to the camera-coordinate averages of the 4 points.

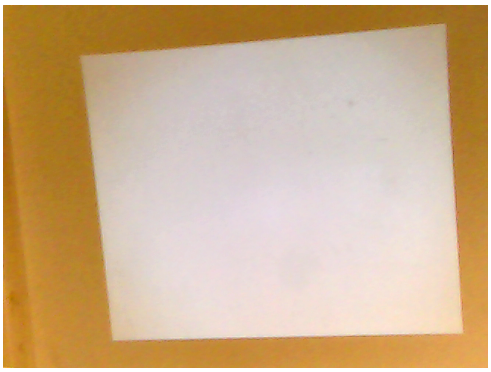
14. **Confirmation:** Draw the detected screen edges and seek user confirmation as to the success of the calibration. If it is denied, repeat from step 2.

15. **Transform:** Create a transformation mapping the screen points to an image of the (optionally downsampled) screen resolution, for later direct comparison with the screen itself.

16. **Light Reference:** If no exposure control is in use, the original background image can be used as the environment lighting reference. Otherwise the exposure should be adjusted as desired (see below), before setting the screen to black and taking a lighting reference image.

Figures 3.3 and 3.4 display the progression of a single calibration attempt.

(a) Background Image

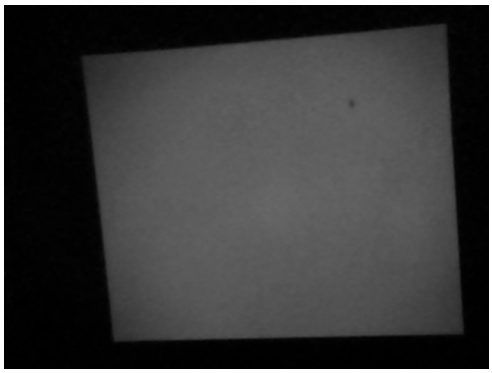


(b) Screen Image

(c) Remove Background

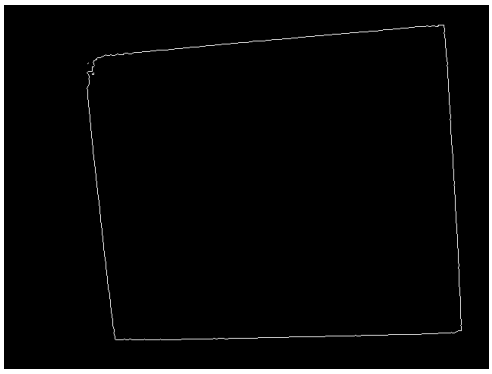
(d) Reduce Noise

Figure 3.3: Screen Calibration Steps (pt 1)

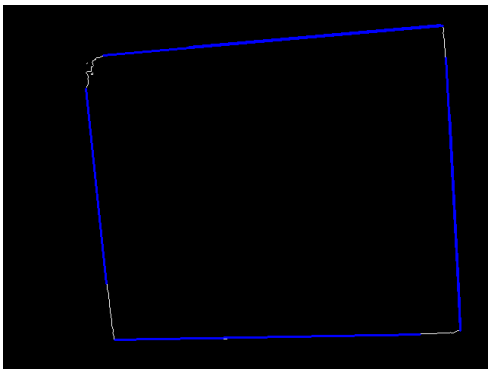


(a) Greyscale

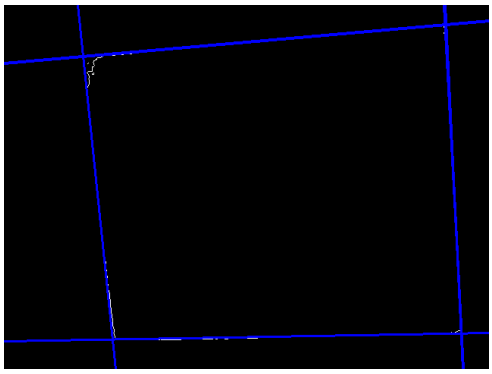
(b) Binarise



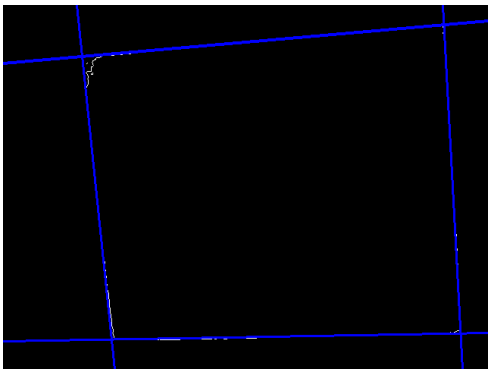
(c) Edges



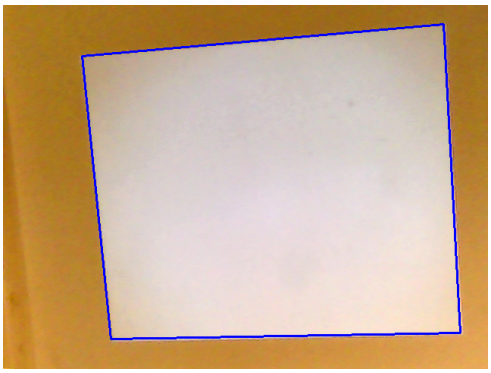
(d) Hough Lines



(e) Scaled Lines



(f) Line Reduction



(g) Calibration Check

Figure 3.4: Screen Calibration Steps (pt 2)

An alternative edge extraction method involving adaptive mean thresholding, morphology, and skeletonisation was implemented but found to be inferior due to being less reliable and more computationally intensive than the current solution.

Characterising the camera with a camera calibration could be helpful, particularly for strongly warped cameras, but was left unimplemented as it is generally non-essential, time-consuming, and involves relatively complicated algorithms. Similar logic applies to performing a colour map from the camera colours to the expected screen colours, with the added drawback of requiring additional computation when processing each image for interaction detection. The main benefit to a colour-mapping approach would occur in cases where the laser pointer is not particularly bright relative to the screen, and needs to be searched for with primarily with colour and shape information.

As suggested in [7], controlling the exposure helps reduce the difficulty of robust interaction detection. This was attempted but not able to be implemented due to difficulties with connecting the available webcams to an appropriate back-end. It is strongly suggested that this is implemented reliably in continuation of this project, particularly for additive-light interactions such as laser-pointers, which may not be detected if the camera is saturated by a high-intensity region of the projection. Where control is established, the exposure should be set to constant (auto-exposure should be turned off), and the exposure should be adjusted such that the projection at maximum intensity still has some sensing range to detect brighter light on top of it. In cases where the projection is quite dim relative to the saturation intensity of the camera, the exposure should be increased to maximise the useful signal range, thus increasing the signal to noise ratio. Note that whenever exposure is controlled, images considered relative to the screen should be intensity-adjusted to account for the known difference in intensity.

3.4 Interaction Detection

Once the screen is correctly detected, the algorithm can focus on detecting interactions. As discussed in Section 2.2, detecting interactions quickly and robustly involves exploiting the understanding of the system and its limitations. This includes intensity, shape, and colour knowledge about the interaction type being detected.

Transforming the camera image to have the screen directly comparable to a screenshot allows for background and lighting removal, which makes distortions on the screen significantly easier to detect. Downsampling and blurring help to reduce high-frequency noise from the sensor, and targeted kernel and colour filtering help to quickly find the desired interaction(s). The following list outlines the implemented steps to process a sensor image and determine the position of an interaction, if one is present. The relevant general interaction and laser detection code is in Appendix E.

1. **Latest Frame:** Retrieve the latest camera frame, blur it slightly with a Gaussian kernel to reduce sensor noise, and transform it to match the shape of the (optionally downsampled) screen.

2. **Reference (Background) Image:** While waiting for step 1., take a screenshot of the source image being sent to the projector, resize appropriately if downsampling, and add on the reference lighting if using.

3. **Background Subtraction:** Once the latest frame and reference image are both available, subtract the reference image from the latest frame to remove the background, and blur with a Gaussian kernel to reduce the effects of high frequency noise from the lighting reference.

4. **Normalisation:** Linearly scale the range of the resulting image such that the minimum value (negative from the subtraction) is scaled to zero, and the maximum value remains where it is. This helps to improve the information density within the valid range of the image.

5. **Greyscale:** Convert the image to a single channel by either converting to a greyscale intensity map, or using known colour information of the interaction to choose an optimal channel (e.g. use the same colour channel as a laser colour).

6. **Seek Interaction Shape:** Convolve the image with one or more custom-shaped kernels intended to identify the known/expected shape of the interaction being searched for (e.g. a laser-pointer can be found using a small kernel with a high peak in the center and sharp drop-off around it).

7. **Remove DC Error:** Subtract the previous filtered image to remove stationary background components accentuated by the filter. This helps to focus on moving interactions.

8. **Blurring:** Blur the image with a broad Gaussian kernel to help reduce noise accentuated by the filter.

9. **Threshold:** Determine interaction location(s) by comparing the highest scored points with a pre-determined threshold. When tracking only one interaction, simply find the highest value point.

10. **Handling:** If not compensating, register any valid points as the latest interaction location(s). Otherwise, register these points as correcting-measurements for the compensation algorithm (see Section 3.5). Scale up point(s) by downsampling factor to get true screen position(s).

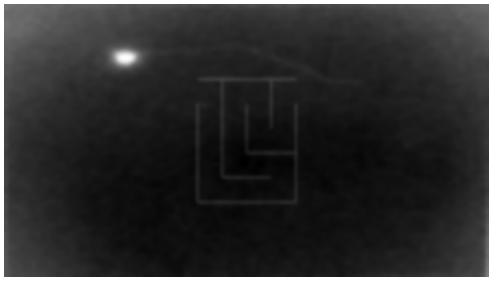
Figures 3.5 and 3.6 display the progression of a single laser detection.

(a) Background Image (b) Camera Image

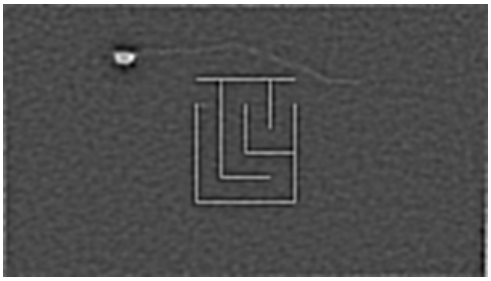
Figure 3.5: Laser Detection Steps (pt 1)

(a) Remove Background

(b) Normalise



(c) Greyscale



(d) Shape Filter

(e) Remove DC Error

(f) Blur

Figure 3.6: Laser Detection Steps (pt 2)

3.5 Compensation

Detection of interactions is not guaranteed to be a fast process, so it can be useful to apply a compensation algorithm to provide a smooth response even when detections are slow. Compensation options and implementation are outlined in this section.

3.5.1 Utility of Compensation

As with any feedback system, there is delay between when the system is in a given state and when the sensors report that state. By the time the system has detected an interaction from the sensor data, the interaction is most likely no longer in that location. Accordingly, it can be beneficial to use a compensation algorithm to predict the expected current location of an interaction, as opposed to simply displaying where it was last recorded. An effective compensation algorithm is able to exploit measurements and time information to determine accurate predictions of the current state of an interaction. A compensator also has the ability to account for missed measurements, in which case brief detection failures do not cause significant issues.

It may at times be beneficial to perform multiple predictions between observations to maintain a desired update frequency, thus reducing jerkiness of the displayed interaction result(s). This, however, comes at the cost of potential overshoot, particularly between slow observations, which could outweigh the benefit of smoother updates. Multiple predictions could also result in larger jerks from correcting for overshoot than those ordinarily between observations. Similarly, it is worth noting that it is possible to begin the next measurement while processing previous measurement(s), thus increasing the temporal resolution of compensator corrections. In such a system, care must be taken to avoid increasing the delay by overloading the processing capacity of the device, as asynchronously processing several measurement attempts simultaneously could cause individual measurements to take longer. In cases of increased delay, the error reduction from finer resolution motion is effectively negated, so it is not worth the compromise.

For the sake of simplicity and ease of implementation, the target system is designed to run sequentially, whereby the processing of each measurement blocks until completion, after which a prediction is made and the next measurement is begun. This helps to reduce overshoot and saves processing power. Avoiding excessive processing also saves energy, which could be particularly relevant in scenarios where the interaction system is being run with power constraints. Nevertheless, it is suggested that asynchronous measurement and prediction be tested in future works to determine the utility of alternative systems.

3.5.2 Compensator Selection

Selection of an appropriate compensator involves determining the desired type of tracking, and understanding the available information and system constraints:

- **No Control Inputs:** User has autonomy and no direct means of communicating with the controller.
- **Measurement Definition:** A single measurement is a single interaction position at a recorded time.
- **Position Components:** Given the user can interact wherever they please, position components of an interaction are independent.
- **Location Uncertainty:** Requires frequent measurements and fast processing to correct for temporal delay between when a measurement is taken and when the corresponding interaction is registered.
- **Desired Compensation:** Follows the true interaction position precisely and accurately across both time and space. Continues through missed measurements, but also stops when interaction stops.
- **Realistic Compensation:** Given the interactions are intended to be meaningful to human users, processing above 30FPS (standard frame-rate of TV) will have limited returns. The minimum required frame-rate is estimated to be around 12FPS in animation [9], below which jerkiness becomes excessive and difficult to follow for human viewers.
- **Comparability:** An uncompensated system can be considered as equivalent to zero-order hold compensation, where each prediction simply returns the last measured state. This allows for both sequential and asynchronous comparisons.

From these properties it can be summarised that the goal of the compensator is to track an interaction point in real-time, accounting for uncertainty of both position and trajectory. Two common methods for object tracking are extended Kalman filters (EKFs) and particle filters (PFs). Both allow for asynchronous predictions and corrections from measurements, while accounting for uncertainty of process, measurement, and state, but a PF uses several samples which converge to one or more most likely locations, whereas an EKF predicts just one location for the point it is tracking [10]. Additional compensation methods such as machine learning techniques are also possible, but the simplicity and rapid computation of an EKF makes it a highly suitable compensation algorithm for the desired use-case.

3.5.3 Compensator Implementation and Tuning

Kalman filters have several variations. The EKF is most commonly used for tracking, as it is augmented with kinematics, effectively allowing one implicitly evaluated time-derivative of state beyond that which is measured. Accordingly, if the interaction detector measures the position of the interaction, kinematic Kalman compensation can make predictions about the current position by assuming the velocity is constant from the previous position. More complex models are possible, accounting for acceleration and/or turn-rate for turning systems, but require additional measurement values, and were not implemented in the target case due to additional implementation complexity, longer computation times, and larger spikes from rapid direction changes if tracking acceleration.

In order for a compensation algorithm to be effective, it must be tuned for the desired use-case. A Kalman filter works by assuming Gaussian uncertainty for the measurements, and the process - the expected changing of the system [11]. Tuning a Kalman filter involves essentially estimating these uncertainty values accurately, hence increasing the likelihood of correct predictions. As the interaction locations are in pixel coordinates, the noise (uncertainty) covariances are also measured in pixel units.

Table 3.1: Tuned noise variance

Noise Type	Variance	Reasoning
Measurement	0.01 pixels	Detected interactions generally very accurate
Process Position	500 pixels	Positions can change significantly between measurements
Process Velocity	10 pixels/second	Velocity can change, but not as quickly as position

Process noise was determined through repeated testing of the compensation. All off-diagonal terms of the noise covariance matrices were set to 0 to reflect the independence of the directions. These values are somewhat arbitrary, but it is worth noting that any interaction type will likely require its own compensation tuning, and different users, devices, and/or cameras would likely benefit from individualised tuning. Accordingly, it is suggested that future work considers a self-tuning approach for the compensation, either through a tuning mode at setup, or continually while using the interaction framework. Appendix E contains the compensation implementation code, adapted with significant modifications from [12].

3.6 Further Considerations

With the compensation implemented, there remain some aspects of the framework that are unimplemented. They are discussed here for completeness.

Intentionally waiting for display updates is only meaningful when there is a particular display image the controller is waiting for. Generally, the display is not manually updated except in testing mode, where a line is drawn from the previous to the current interaction. In the target implementation interactions are registered as an influence on the source, but no attempt is made to predict their effect, and instead the algorithm works purely on what is detected by the sensor, and what is on the screen when it is captured.

In the example implementation only one interaction type is being tracked, so `pyautogui` was used to simply move the laptop mouse cursor to the detected laser position. Tracking multiple interactions while emulating a trackpad rather than a standard mouse would allow for gestures such as zooming and rotating. Advanced interaction detection could also be used to recognise gestures of a single interaction over time, allowing for actions such as clicking and dragging.

Presently the system is not set up to handle changing screen resolutions post-calibration. As Microsoft Powerpoint often does this when entering presentation mode, it is suggested to either set a consistent resolution, or start a presentation prior to running the interaction detector, then switching back to the presentation once calibration is complete.

Chapter 4

Results/Analysis

With the system implemented, it is possible to test the success of the algorithm in terms of its speed and the benefit of the compensation. Testing was performed by an expert user of the system and two novices, with instructions to draw a line with the laser pointer through a simple maze as quickly as possible. This was done several times with and without compensation applied, and over varying added delays to simulate the effects of a slower system. A sample trial is displayed in Figure 4.1. The attempts were recorded using testing mode of the control algorithm (Appendix A), and separated and summarised using the code in Appendix F.

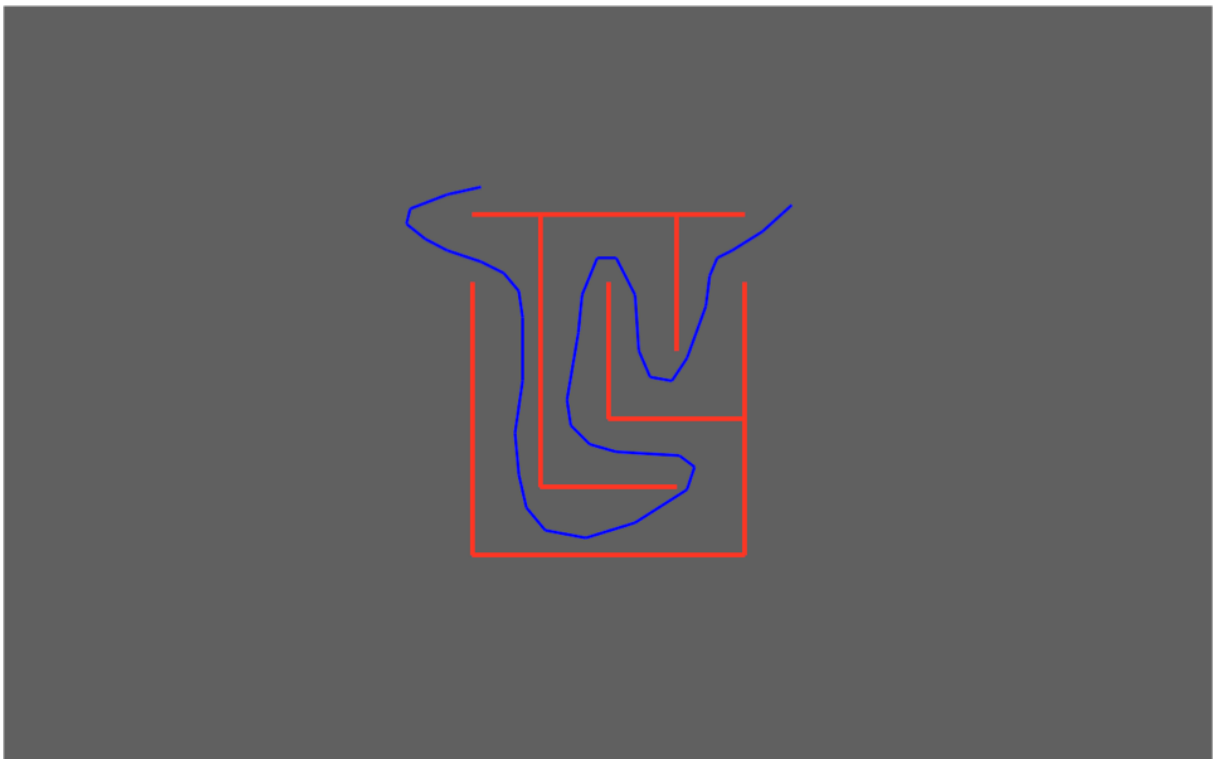


Figure 4.1: Maze Sample Trial

4.1 Sufficiency

Within the testing conditions, screen and laser detection were highly robust (see Figure 4.2 for an example screen detection), so the main determining factor of sufficiency was processing speed.

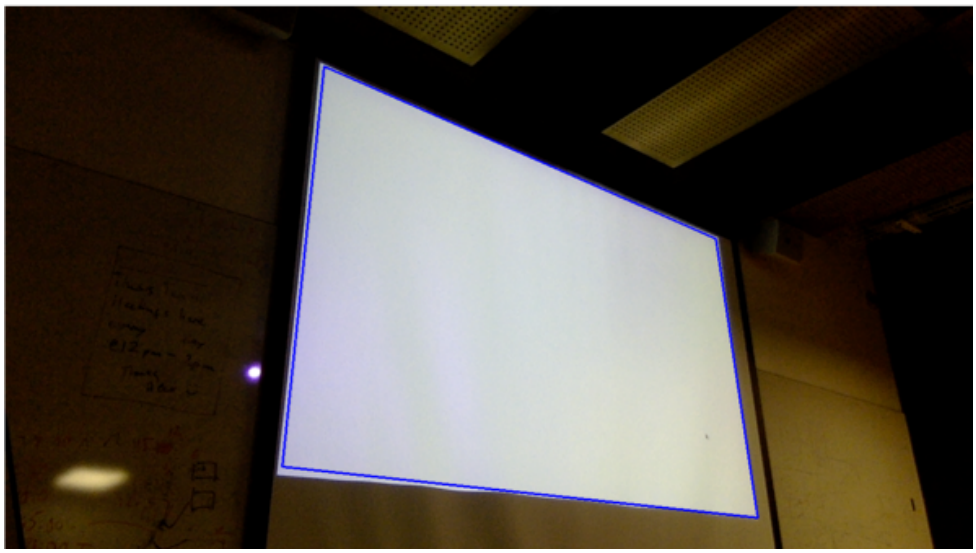


Figure 4.2: Screen Detection Example

Figure 4.3 displays the effective processing frame-rate for the base system, as well as for added delays of 10, 50, and 100ms. The variance of the processing times is considerably larger than the added computation time when using the compensated system, so compensation is assumed to add negligible delay.

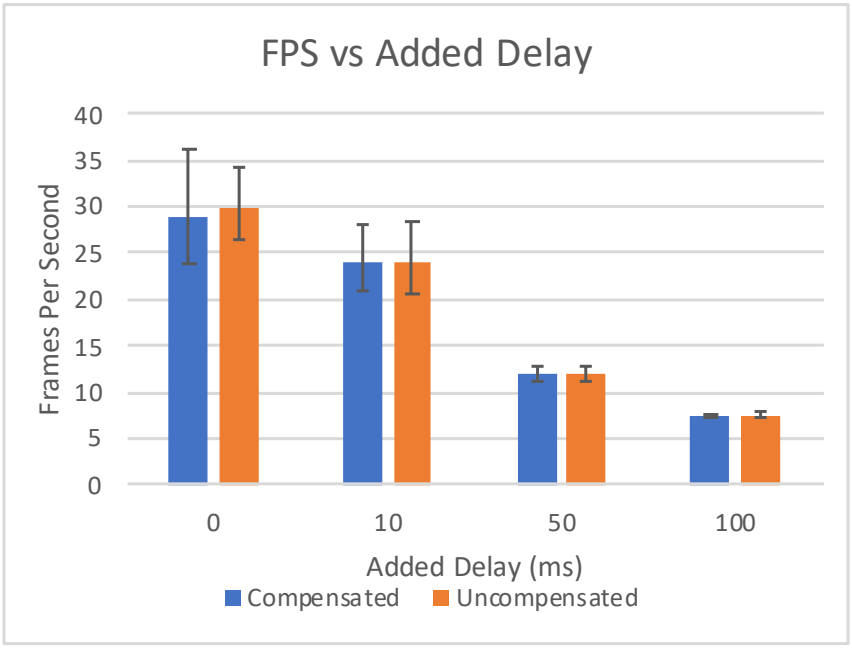


Figure 4.3: Equivalent Frames Per Second Processing Speed

The general consensus of the three testers was that the system worked well for no added delay, and 10ms of delay. At 50ms added delay it was considered usable but not ideal.

100ms of added delay was sufficiently jerky that the system would likely be more of a hindrance than a benefit in a presentation environment. Timing results from the successful attempts through the maze are presented in Figure 4.4, with times normalised to the fastest expert time. Due to the nature of the test instructions, only the fastest attempt time was plotted for each category, as the test was one of high-fidelity tracking at maximum speed. The slower completion times for both novices using the base system are attributed to this being the first test performed, after which they were more used to the system and instructions. This should be controlled for in future trials.

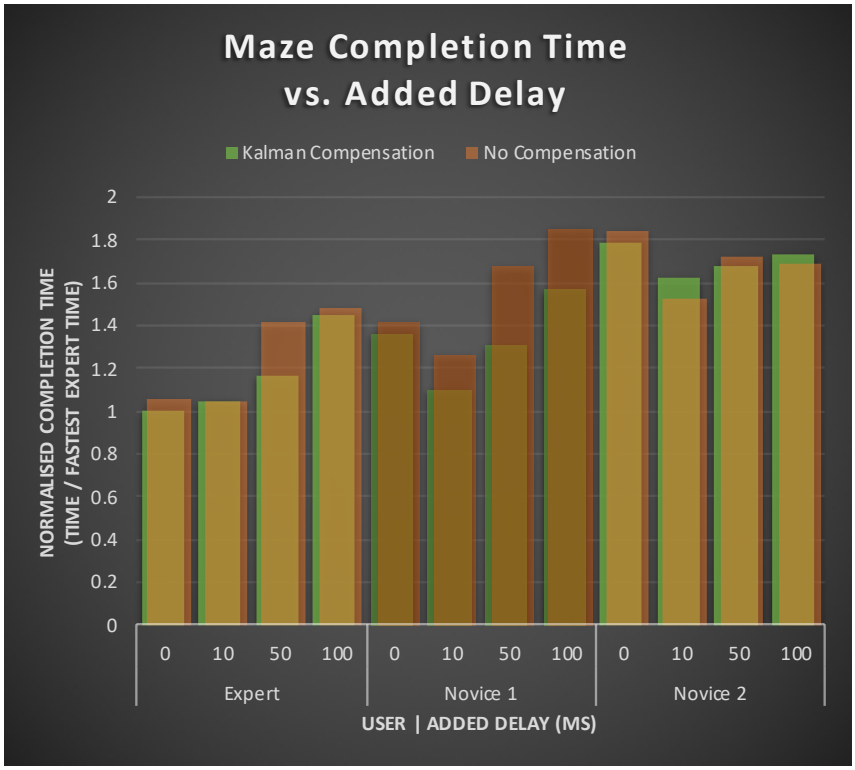


Figure 4.4: Effect of Added Delay on Usability (Fidelity at Speed)

With the base system processing each frame within about 30ms, adding 50ms of delay brings the frame rate to approximately 12.5FPS, which is consistent with the 12FPS lower limit of reasonableness specified in Section 3.5.2. Accordingly, the base system was deemed sufficient, and could run on computers up to 2.67 times slower than the test device while maintaining usability.

4.2 Baseline Comparison

While Section 4.1 determines the compensation evaluates fast enough to not be detrimental to performance, it is worth considering whether or not it is actually helpful. Figure 4.5 extracts the ratio of uncompensated and Kalman-compensated completion times from Figure 4.4. Here it can be seen that in general Kalman compensation was helpful in reducing maze completion time, with uncompensated times up to 20% slower for some tests. It is suggested to perform additional tests in future to determine the relationship between compensation and added delay, using a larger test population, and to determine the effect of tuning the compensation algorithm specifically to maximise performance at each delay amount, or even specifically for each user.

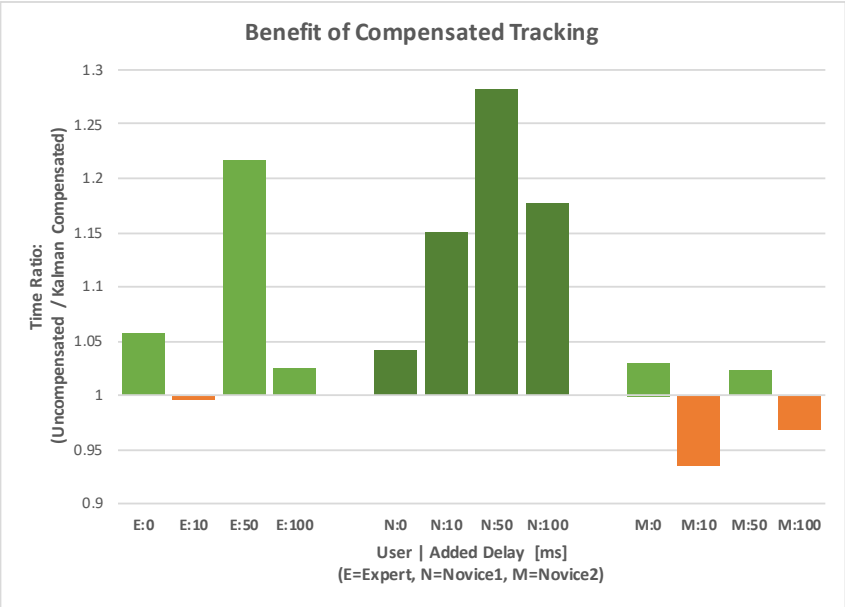


Figure 4.5: Compensated Tracking Maze Completion Time Benefit

Chapter 5

Conclusion and Recommendations

A need for cost-effective, scalable, and transportable projection interaction technology has been identified, which can facilitate dynamic, interactive presentations. To this end, an open-source framework for detecting and managing interactions has been developed, with the target implementation of Kalman-compensated laser-pointer detection implemented successfully using a webcam.

A review of relevant literature found existing techniques for screen and laser-pointer detection within an image were documented, but not optimal for the use-case where the controller can exploit temporal control and the known background to more robustly perform detection. Reported techniques in the reviewed articles also did not provide accessible code, so the open source community was unable to use their results without implementing them themselves.

Accordingly, a framework was developed covering screen detection, laser-pointer detection, Kalman compensation, and handling of detected interactions into mouse movements (code in Appendices). The framework is modular and expandable, allowing for easy implementation of additional interaction detection methods, and compensation methods. This system was shown to be usable in testing, for computers up to 2.67 times slower than the test device.

For full utility of the program, it is suggested that exposure control be implemented, with additional potential benefit from implementing camera calibration to de-warp images, and colour-mapping to more easily remove the expected screen from camera images.

Further to these refinements, some other possible extensions are:

- Handling changing screen resolution
- Automatic tuning of the compensation algorithm either during setup or throughout usage by a presenter.
- Asynchronous predictions and measurements to reduce movement jerkiness, and allow for more frequent updates.

Bibliography

[1] SMART Technologies, "Understanding DViT (Digital Vision Touch) Technology," 2019. [Online]. Available: https://community.smarttech.com/servlet/fileField?entityId=ka40P000000H4QwQAK&field=Attachment__Body__s

[2] ——, "Understanding DIR (Distributed Infrared) Technology," 2018. [Online]. Available: https://community.smarttech.com/servlet/fileField?entityId=ka40P000000LFQHQA4&field=Attachment__Body__s

[3] ——, "Understanding HyPr Touch Technology," 2018. [Online]. Available: https://community.smarttech.com/servlet/fileField?entityId=ka40P000000LFQFQA4&field=Attachment__Body__s

[4] A. Wilson and H. Benko, "Combining multiple depth cameras and projectors for interactions on, above and between surfaces," in *Proceedings of the 23nd annual ACM symposium on user interface software and technology*, ser. UIST '10. ACM, 2010, pp. 273–282.

[5] L.-R. Dung, G.-Y. Lai, and Y.-Y. Wu, "Shadow touching for interactive projectors," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 1798–1802.

[6] H. A. Le, T. Doan, and M.-T. Tran, "Webcam-Based Laser Dot Detection Technique in Computer Remote Control," *Journal of Science and Technology (Vietnam)*, vol. 48, 01 2012.

[7] B. Ahlborn, D. Thompson, O. Kreylos, B. Hamann, and O. Staadt, "A practical system for laser pointer interaction on large displays," in *Proceedings of the ACM symposium on virtual reality software and technology*, ser. VRST '05. ACM, 2005, pp. 106–109.

[8] D. Olsen Jr and T. Nielsen, "Laser pointer interaction," in *Conference on Human Factors in Computing Systems: Proceedings of the SIGCHI conference on Human factors in computing systems*, 2001, pp. 17–22. [Online]. Available: http://search.proquest.com/docview/31252314/

[9] T. Sito, *Timing for Animation*. Routledge, apr 2013. [Online]. Available: https://www.taylorfrancis.com/books/9780080951720

[10] M. F. Aydogmus, O.; Talu, "Comparison of Extended-Kalman- and Particle-Filter-Based Sensorless Speed Control," *IEEE Transactions on Instrumentation and Measurement*, vol. 61, no. 2, pp. 402–410, 2012.

[11] T. Babb, *How a Kalman filter works, in pictures*, Bzarg, Aug. 2015. [Online]. Available: https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/

[12] W. Lucetti, *[Tutorial OpenCV] Ball Tracker using Kalman Filter*, Jun. 2015. [Online]. Available: https://www.myzhar.com/blog/tutorials/tutorial-opencv-ball-tracker-using-kalman-filter/

Appendix A

Controller Code

Listing 1: Code enabling user interactions to influence the source

```
#!/usr/bin/env python3

from GUI import *                          # GUI for calibration/testing
                                           #   np, cv2, Edge, device, sleep
                                           #   exit_with_message
from laser_detector import LaserDetector # interaction detector
from threading import Thread, Lock         # semi-synchronous parallel processing
from queue import Queue                    # mouse position thread communication
from time import time                      # timing
from mss import mss                        # fast cross-platform screenshots
from MyImgLib import imadjust              # image value scaling
from analyser import Analyser              # test mode analysis

class Controller(object):
    ''' A class for interfacing a screen with a laser pointer. '''

    COLOUR = {'r': 2, 'g': 1}

    def __init__(self, laser_colour='r', grayscale=False, light_ref=True,
                 compensation=True, user='', delay=0, testing=False):
        ''' Create a controller instance for the screen. '''
        self._cam = None
        self._colour = self.COLOUR[laser_colour]
        self._laser_colour = laser_colour
        self._detector = LaserDetector(laser_colour, kalman=compensation)
        self._GUI = None
        self._mouse_offset = np.array([10,10], dtype=np.float64)

        self._points = []

        self.grayscale = grayscale
        self.light_ref = light_ref

        if testing:
```

```
            filename = 'temp/stats_d' + str(delay)
            if user:
                filename += '_'+user
            if compensation:
                filename += '_c'
            self._filename = filename

        self._testing = testing

        self._delay = delay

        device.PAUSE = 0 # no need for safety delay

    def run(self, camera_id, downsample=4):
        ''' Link the specified camera to the screen and run test maze. '''
        self._queue = Queue()
        device_controller = Thread(name='device', target=self._move_mouse,
                                   daemon=True)
        self._mouse_gain = np.array([1,1], dtype=np.float64)
        self._res_changed = False

        try:
            connect_success = self._connect(camera_id)
            if not connect_success:
                exit_with_message("Connection Failure.")

            # start tkinter (it dies if opened after imshow in _calib_wait)
            self._GUI = GUI()

            # wait for user to confirm connection or quit
            if not self._calib_wait():
                exit_with_message('User quit application.')

            calibrate_success, self._screen = self._GUI.calibrate(self._cam,
                                                                   downsample)
            if not calibrate_success:
                exit_with_message('Calibration failure.')

            # set mouse gain so positions are over full screen
            self._mouse_gain *= self._screen.downsample

            # set boundaries to keep points inside
            self._mouse_offset /= 2
            maxs = self._GUI.resolution - self._mouse_offset
            mins = self._mouse_offset
            self._boundaries = [mins[0], maxs[0], mins[1], maxs[1]]
            self._mouse_offset[:] = [0,0]

```

```
            self._boundary_lines = Edge.get_rect_lines(self._boundaries)

            if self.light_ref:
                self._set_ref_lighting()

            if self._testing:
                self._GUI.draw_maze(self._laser_colour)
                self._points.append([*self._GUI._prev, time()])
            else:
                self._GUI._root.withdraw()
                self._points = [device.position()]

            device_controller.start()  # start mouse controller

            self._start_camera_handler()

            self._latest = 0
            self._last = time()
            if self._testing:
                self._GUI._root.after(self._delay, self._run_loop)
                self._GUI._root.mainloop()
            else:
                while True: self._run_loop()

        finally:
            if self._testing and len(self._points) > 2:
                #print(self._points)
                an = Analyser(self._points, self._GUI._maze_regions,
                         self._GUI._lines)
                print(min(an._dTs), max(an._dTs), np.mean(an._dTs))
                filename = an.save_stats(self._filename)
                print('SAVED STATS TO', filename)

            if not self._cam is None:
                self._cam.release() # release the camera
            try:
                self._GUI._root.destroy()
            except Exception: pass  # root already destroyed/never initialised
            cv2.destroyAllWindows() # stop displaying stuff
            device.mouseUp()        # release the mouse
            # device_controller and frame_thread daemons auto-destroyed

    def _start_camera_handler(self):
        ''' Starts the handler thread for the feedback camera. '''
        self._lock1 = Lock()
        self._lock2 = Lock()
        self._camera_handler = Thread(name='frame', target=self._handle_camera,
                                      daemon=True)
```

```
131         self._lock1.acquire()       # start with main thread in control
132         self._camera_handler.start() # start the camera handling thread
133 
134     def _handle_camera(self):
135         ''' Captures and pre-processes latest image from the camera stream. '''
136         while "running":
137             self._wait_until_needed()
138             # read the latest image from the camera
139             read_success, frame = self._cam.read()
140             if not read_success:
141                 raise Exception("Can't receive frame")
142 
143             # perform initial processing on the image
144             if self.grayscale:
145                 frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
146             tformed = self._screen.transform(frame)
147             self._frame = cv2.GaussianBlur(tformed, (5,5), 0).astype(np.float32)
148 
149             self._inform_image_ready()
150 
151     def _wait_until_needed(self):
152         ''' Wait for main to request the next image. '''
153         self._lock2.acquire() # wait until previous image has been received
154         self._lock1.acquire() # wait until next image is desired
155         self._lock2.release() # inform camera is taking image
156 
157     def _inform_image_ready(self):
158         ''' Inform main that next image is available. '''
159         self._lock1.release() # inform next image is ready
160 
161     def _get_latest_image(self):
162         ''' Ask camera handler for next image. '''
163         self._lock1.release() # inform next image is desired
164         self._lock2.acquire() # wait until camera is taking image
165 
166     def _wait_for_camera_image(self):
167         ''' Wait until next image is available. '''
168         self._lock1.acquire() # wait until next image is ready
169         self._lock2.release() # inform image has been received
170 
171     def _run_loop(self, debug=False):
172         ''' Internal loop for updating mouse position. '''
173         # retrieve the latest image in a separate thread
174         self._get_latest_image()
175         # while generating a reference image in this one
176         ref_img = self._get_ref_img()
177         self._wait_for_camera_image()
178         # combine latest image with its reference
```

```
        removed_bg = cv2.GaussianBlur(self._frame - ref_img, (5,5), 0)
        # imadjust (negative) min to 0
        process = imadjust(removed_bg, high_out=removed_bg.max())

        # only ever used manually
        if debug:
            cv2.imshow('new',new.astype(np.uint8))
            cv2.imshow('ref',imadjust(ref_img).astype(np.uint8))
            cv2.imshow('light',self._ref_light.astype(np.uint8))
            cv2.imshow('process',process.astype(np.uint8))
            print('n:',new.min(),new.max(),'r:',ref_img.min(),ref_img.max(),
                  'l:',self._ref_light.min(),self._ref_light.max(),
                  'p:',process.min(),process.max())

        # make and register a measurement
        self._detector.detect_interaction(process)
        self._predict() # estimate the current position

        if self._testing:
            # run this function again after self._delay ms
            self._GUI._root.after(self._delay, self._run_loop)
        # else loop run externally to avoid overloading call stack

    def _predict(self):
        ''' Makes a prediction - if valid moves mouse there. '''
        position = self._detector.predict()
        if position is not None:
            position = position.reshape(-1)[:2]
            # shift mouse position into valid range (scale to shift points from
            #   downsampled positions, and offset to avoid edges)
            mouse_pos = self._position_to_mouse(position)

            if mouse_pos is None:
                return

            # put the new position in the mouse-move queue
            self._queue.put(mouse_pos)
            if self._testing:
                # track the position for analysis
                self._points.append([*mouse_pos, time()])
                # draw the new position on the canvas
                self._GUI.draw_line(*mouse_pos)
            else:
                self._points = [mouse_pos]

    def _position_to_mouse(self, position):
        ''' Returns a valid mouse_position given the latest predicted position.
```

```
227         Ensures points are within the desired boundary
228 
229         self._position_to_mouse(np.arr[float(x2)]) -> np.arr[float(x2)]
230 
231         '''
232         mouse_pos = position * self._mouse_gain + self._mouse_offset
233         if Edge.is_out_of_bounds(mouse_pos, self._boundaries):
234             _, mouse_pos = Edge.get_path_interesection(
235                 [*mouse_pos, *self._points[-1][:2]], self._boundary_lines,
236                 position=True)
237         return mouse_pos
238 
239     def _connect(self, camera_id):
240         ''' Connect to the specified camera. '''
241         first = True
242         while first or not self._cam.isOpened():
243             if first:
244                 # use expected camera id this time
245                 first = False
246             else:
247                 print('Invalid Camera ID:', camera_id)
248                 camera_id = input('Camera ID: ')
249 
250             self._cam = cv2.VideoCapture(camera_id)
251 
252             if cv2.waitKey(1) == ord('q'):
253                 print('Application quit by user')
254                 return False
255 
256         return True
257 
258     def _calib_wait(self):
259         ''' Wait for the user to begin the calibration. '''
260         print('press c to begin calibration')
261         while 'not started':
262             key = cv2.waitKey(1)
263             if key == ord('c'):
264                 break # waiting done
265             elif key == ord('q'):
266                 return False # user quit
267 
268             read_success, frame = self._cam.read()
269             if read_success:
270                 cv2.imshow('calibration', frame)
271             else:
272                 print('failure')
273 
274         return True
```

```
275
276     def _set_ref_lighting(self):
277         ''' Set the reference image for lighting subtraction. '''
278         self._GUI._fullscreen_gui()
279         # make the screen black to see background lighting
280         self._GUI._root.config(bg='black')
281         self._GUI._update_projection()
282         read_success, ref_img = self._cam.read()
283         if read_success:
284             self._ref_light = cv2.GaussianBlur(self._screen.transform(ref_img),
285                                                (5,5), 0).astype(np.float32)
286         else:
287             raise Exception('Lighting reference image unable to be taken -'\
288                             'check camera connection')
289
290     def _get_ref_img(self):
291         ''' Get the latest reference image using the screen and lighting. '''
292         width, height = self._screen.resolution * self._screen.downsample
293         monitor = {'top':0, 'left':0, 'width':width, 'height':height}
294         with mss() as sct:
295             #mouse_pos = device.position()
296             img = np.array(sct.grab(monitor), dtype=np.float32)
297
298         #self._add_cursor(img, *mouse_pos)
299
300         # convert colour as appropriate
301         if self.grayscale and not self.light_ref:
302             img = cv2.cvtColor(img, cv2.COLOR_BGRA2GRAY)
303         else:
304             img = cv2.cvtColor(img, cv2.COLOR_BGRA2BGR)
305
306         # resize with automatic antialiasing low pass filtering
307         fx, fy = self._screen.resolution / img.shape[1::-1]
308         # only re-size if actually changing size
309         if fx != 1 or fy != 1:
310             img = cv2.resize(img, (0,0), fx=fx, fy=fx,
311                              interpolation=cv2.INTER_AREA)
312         if self.light_ref:
313             img += self._ref_light
314             if self.grayscale:
315                 img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
316         return img
317
318     def _add_cursor(self, img, x, y):
319         ''' Adds the cursor to the provided screenshot image. '''
320         # crop cursor image to witihin the screen
321         x_subi, y_subi, x_addi, y_addi = self._GUI.cursor_info
322         x_sub = min(x_subi, x)
```

```
        y_sub = min(y_subi, y)
        ym, xm = img.shape[:2]
        xp, yp = xm - x, ym - y
        x_add = min(x_addi, xp)
        y_add = min(y_addi, yp)

        cursor_img = self._GUI.cursor_img[y_subi-y_sub:y_subi+y_add,
                                          x_subi-x_sub:x_subi+x_add]
        # add cursor to img using complementary alpha-channel information
        alpha = np.repeat((cursor_img[:,:,3] / 255.0).reshape(
            *cursor_img.shape[:2], 1), 3, axis=2)

        img[y-y_sub:y+y_add, x-x_sub:x+x_add, :3] = alpha*cursor_img[:,:,:3] + \
                (1.0 - alpha) * img[y-y_sub:y+y_add, x-x_sub:x+x_add, :3]

    def _move_mouse(self):
        ''' Receives continual instructions from the main thread for where to
            move the mouse.
        '''
        while 'still useful':
            device.moveTo(*self._queue.get())

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(
        description='control the mouse using a laser-pointer')
    parser.add_argument('-NC', '--noCompensation', action='store_true',
                        help='disable Kalman compensation')
    parser.add_argument('-i', '--cameraID', default=0, help='camera ID',
                        type=int)
    parser.add_argument('-c', '--colour', help='laser colour', default='g',
                        choices=['g', 'r'])
    parser.add_argument('-d', '--downsample', help='downsample screen by',
                        type=int, default=4)
    parser.add_argument('-NL', '--noLightRef', action='store_true',
                        help='disable removing background lighting')
    parser.add_argument('-t', '--testing', help='use test mode',
                        action='store_true')
    parser.add_argument('-u', '--user', help='the user testing the system',
                        default='me')
    parser.add_argument('-D', '--delay', default=0, type=int, help='ms delay'+\
                        ' added between processing frames while testing')
    args = parser.parse_args()

    C = Controller(args.colour, compensation=not args.noCompensation,
                   light_ref=not args.noLightRef, user=args.user,
                   delay=args.delay, testing=args.testing)
    C.run(args.cameraID, args.downsample)
```

Appendix B

Calibration and Testing Code

Listing 2: Code for calibrating and testing the interaction detection system

```
#!/usr/bin/env python3

from sys import platform # OS handling
from time import sleep   # efficient waiting
from screen import *     # Screen, np, cv2, device, Edge
import tkinter as tk     # GUI components

class GUI(object):
    ''' A class for managing the GUI used for calibration and testing. '''
    def __init__(self):
        ''' Create the GUI instance. '''
        self._root = tk.Tk()
        self._root.config(bg='black')
    def _fullscreen_gui(self):
        ''' Make the GUI full-screen, and bring into focus. '''
        self._root.deiconify()
        self._root.attributes('-fullscreen', True)
        self._canvas.focus_force() # bring into focus

    def _create_canvas(self):
        ''' Create the canvas element for display control. '''
        width, height = self.resolution = Screen.get_resolution()
        self._canvas = tk.Canvas(master=self._root, width=width, height=height,
                                 bg='black', highlightthickness=0,
                                 cursor='none')#self._cursor)

        # set up binds to enable user to quit
        self._canvas.bind('<Key>', self.key)
        self._canvas.bind('<Escape>', lambda e: exit_with_message())

        self._canvas.pack(fill=tk.BOTH, expand=True) # fill space available
        self._canvas_lines = []

    def _calibration_setup(self):
```

```
        ''' Perform relevant setup for the calibration. '''
        # ensure full-screen works
        self._root.overrideredirect(True)
        self._root.overrideredirect(False)

        self._create_canvas()

        # create a resizable window for displaying calibration results
        cv2.namedWindow('calibration', cv2.WINDOW_NORMAL)
        cv2.moveWindow('calibration', 0, 20)

    def calibrate(self, cam, downsample):
        ''' Calibrate the specified camera using the GUI as a known screen. '''
        print('Beginning Calibration')
        self._calibration_setup()

        while 'not calibrated':
            print('attempting to calibrate')
            self._fullscreen_gui()

            # take a background image
            self._canvas.config(bg='black')
            self._update_projection()
            bg_success, bg = cam.read()
            bg = bg.astype(np.int64) # allow for negatives in subtraction

            # take an image with the screen white
            self._canvas.config(bg='white')
            self._update_projection()
            frame_success, frame = cam.read()

            if not (frame_success or bg_success):
                print("Can't receive frame (stream end?)")
                return False, None

            try:
                image = frame - bg
                image[image < 0] = 0
                points = Screen.points_from_img(np.uint8(image), colour='W')
            except Exception as e:
                print('calibration failed:', e)
                continue

            screen = Screen(points, resolution=self.resolution,
                            downsample=downsample)

            # adjust the calibration to improve accuracy
            try:
```

```
                key = self._display_calibration(frame, screen)
                if key == ord('c'):
                    cv2.destroyWindow('calibration') # cleanup
                    return True, screen
                elif key == ord('q'):
                    exit_with_message('User quit application')
            except Exception as e:
                print('Adjusting failed. Retrying...')
                continue
    def _display_calibration(self, display_frame, screen):
        ''' Displays the detected screen location on the display_frame. '''
        self._root.withdraw()
        cv2.polylines(display_frame, np.int32([screen._points]), True,
                      (255,0,0), 2)
        cv2.imshow('calibration', display_frame)
        return cv2.waitKey(0)

    def draw_maze(self, laser_colour):
        ''' Draw a maze on the screen for testing purposes. '''
        self._fullscreen_gui()
        self._root.config(bg='black')
        width, height = self.resolution
        self._canvas.config(bg='#333333')

        self._lines = np.array([[0,0,4,0],
                                [0,1,0,5],
                                [0,5,4,5],
                                [1,0,1,4],
                                [1,4,3,4],
                                [2,1,2,3],
                                [2,3,4,3],
                                [3,0,3,2],
                                [4,1,4,5]])
        scale = 0.9 * min(width, height)/10
        offset = (np.array([width, height]) - (np.array([4,5]) * scale)) / 2
        offset = np.array(list(offset) * 2)
        self._lines = list(self._lines * scale + offset)
        self._maze_scale = scale

        self._maze_regions = {'start_':[0,1,0,1],
                              't0_1':[0,1,1,4],
                              't90_1':[0,1,4,5],
                              't0_2':[1,3,4,5],
                              't180_1':[3,4,3,5],
                              't0_3':[2,3,3,4],
                              't90_2':[1,2,3,4],
                              't0_4':[1,2,1,3],
                              't180_2':[1,3,0,1],
```

```
                            't0_5':[2,3,1,2],
                            't180_3':[2,4,2,3],
                            't0_6':[3,4,1,2],
                            'end_':[3,4,0,1]}
        offset = offset.reshape((2,2)).T.reshape(-1)
        for region in self._maze_regions:
            self._maze_regions[region] = \
                    list(np.array(self._maze_regions[region]) * scale + offset)

        if laser_colour == 'r':
            fill = 'green'
        else:
            fill = 'red'

        for line in self._lines:
            self._canvas.create_line(*line, width=5, fill=fill)
        self._update_projection()

        device.moveTo(20,20)
        self._prev = [20,20]

    def draw_line(self, mouse_pos, y=None):
        ''' Draw a line following the mouse position. (callback function) '''
        if not y is None:
            new = [mouse_pos, y] # [x,y]
        else:
            new = [mouse_pos.x, mouse_pos.y]
        self._canvas_lines.append(self._canvas.create_line(*(self._prev + new),
                width=3, fill='blue'))

        if len(self._canvas_lines) > 20:
            self._canvas.delete(self._canvas_lines.pop(0))

        self._prev = new
        self._root.update()

    def key(self, event):
        ''' Key-press callback: quit on 'q'. '''
        #print('key event:', event)
        if event.char == 'q':
            self._root.destroy()
            sleep(0.3)
            exit_with_message()

    def _update_projection(self, wait=0.5):
        ''' Wait for the projector to update. '''
        self._root.update()
        for i in range(5):
```

```
179             sleep(wait/5)
180             self._root.update()
181
182
183 def exit_with_message(msg='User quit application.', retval=0):
184     ''' Prints message and exits with code retval. '''
185     print(msg + ' Exiting...')
186     exit(retval)
```

Appendix C

Screen Detection Code

Listing 3: Screen Detection Implementation

```
#!/usr/bin/env python3

import cv2
import numpy as np
from Edge2 import Edge
import pyautogui as device # used for screen resolution

class Screen(object):
    ''' A class for tracking a physical screen display. '''
    def __init__(self, image_points, resolution=None, downsample=4):
        ''' Creates a Screen object from 'image_points'.

        'points' is a list of points in clockwise order from the top left
            e.g. [[top_left], [top_right], [bottom_right], [bottom_left]].
        'resolution' is the resolution of the screen, used for mapping.
            If left as None, resolution is determined as the computer's
            resolution.
        'downsample' is the amount to downsample the screen's resolution by
            in the transform from image to screen. Defaults to 4 if not set.

        Constructor: Screen(np.arr[arr[int(x2)](x4)], [int,int])

        '''
        self.downsample = downsample
        self._points = np.array(image_points, dtype=np.float32)

        # automatically determine (downsampled) screen resolution
        if resolution is None:
            resolution = self.get_resolution(downsample)
        elif downsample is not None:
            resolution = resolution // downsample # replace w/ downsampled copy
        self.resolution = resolution

        width, height = np.array(resolution) - 1
```

```
        self._dst_points = np.array([[0, 0], [width, 0],
                                     [width, height], [0, height]],
                                     dtype=np.float32)
        self._inverse_transform = np.matrix(cv2.getPerspectiveTransform(
            self._dst_points, self._points))

    def transform(self, img, display=False):
        ''' Returns the transformation result, from sampling 'img' using
            the internal transform, which maps to a downsampled view of the
            screen.
        '''
        # transform already inverted, so no need for warpPerspective to invert
        transformed = cv2.warpPerspective(img, self._inverse_transform,
                tuple(self.resolution), flags=cv2.INTER_LINEAR+\
                cv2.WARP_FILL_OUTLIERS+cv2.WARP_INVERSE_MAP)

        if display:
            cv2.imshow('transformed', transformed)
            print('press any key to continue')
            cv2.waitKey(0)

        return transformed

    def recalibrate(self, measured_pts, expected_pts):
        ''' Adjusts the stored defining points and transform using the mapping
            between the measured and expected points.

            'measured_pts' should be from an image pre-transformed with the
                current screen transform.

        '''
        adjustment = cv2.getPerspectiveTransform(np.float32(expected_pts),
                                                 np.float32(measured_pts))
        self._inverse_transform *= adjustment
        self._points = self.transform_pts(self._dst_points,
                                          self._inverse_transform)

    @staticmethod
    def transform_pts(pts, M):
        ''' Returns 2D pts transformed by transformation matrix M.

        Screen.transform_pts(np.arr[[float(x2)](xN)],
                             np.matrix[[float(x3)](x3)])
                -> np.arr[[float(x2)](xN)]

        '''
        in_pts = np.ones((3, len(pts)))
        in_pts[:2] = pts.T
```

```
        out_pts = M * in_pts
        out_pts = (out_pts[:2] / out_pts[2]).T
        return out_pts

    @classmethod
    def points_from_img(cls, img, colour=None, display=False):
        ''' Returns the corner points for a detected screen in 'img'.

        'img' is an image as returned from cv2.imread.
        'colour' is the colour of the screen being detected ('R','G','B','W').
            If left as None, the screen is assumed to be white.
        'display' is a boolean flag specifying if the detected screen corners
            should be displayed on 'img'.

        cls.points_from_img(np.arr[arr[int](x3)], str)
                -> np.arr[arr[int(x2)](x4)]

        '''
        blur = cv2.GaussianBlur(img, (5,5), 0)
        if len(img.shape) == 3 and img.shape[2] == 4:
            blur = cv2.cvtColor(blur, cv2.COLOR_BGRA2BGR)

        if len(img.shape) < 3 or img.shape[2] == 1:
            grey = img
        elif colour is None or colour == 'W':
            grey = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
        else:
            B,G,R = cv2.split(blur)
            if colour == 'R':
                grey = R
            elif colour == 'B':
                grey = B
            elif colour == 'G':
                grey = G
            else:
                raise Exception(('Invalid colour: '+ str(colour)))

        edges = cls._process(grey)
        lines = cls._get_screen_lines(edges)

        if display: # display detected edges
            edges2 = cv2.merge([edges]*3) # convert to colour image
            for line in lines:
                # add lines to image
                p1, p2 = np.array(line.reshape((2,2)), dtype=int)
                cv2.line(edges2, tuple(p1), tuple(p2), (255,0,0), 2)
            cv2.imshow('lines', edges2)
            if cv2.waitKey(0) == ord('q'): # wait for user to continue or quit
```

```
131                 exit()
132
133         points = cls._get_screen_points(lines)
134
135         if display and points: # display detected points and screen polygon
136             cv2.polylines(blur,np.int32([points]),True,(0,0,255),2)
137             for point in points.T:
138                 blur[point[0],point[1]] = [0,0,255]
139             cv2.imshow('lines', blur)
140             cv2.waitKey(1) # show to the display
141         return points
142
143     @classmethod
144     def _process(cls, grey, canny_lower=0, canny_upper=100, canny_aperture=3):
145         ''' Performs pre-processing on the provided greyscale image.
146
147         cls._process(np.arr[arr[int]]) -> np.arr[arr[int]]
148
149         '''
150         # Otsu binarisation (threshold assuming bimodal intensity histogram)
151         th, bw = cv2.threshold(grey,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
152         # Canny edge detection
153         edges = cv2.Canny(bw, canny_lower, canny_upper,
154                           apertureSize=canny_aperture)
155         return edges
156
157     @staticmethod
158     def _get_screen_lines(edges, col_thresh=5, dist_thresh=0):
159         ''' Gets the bounding lines of the screen in the 'edges' image.
160
161         'col_thresh' is a collinearity threshold (angle in degrees) for
162             combining similar lines in the image.
163         'dist_thresh' is a minimum distance threshold (in pixels) for combining
164             similar lines in the image.
165
166         Screen._get_screen_lines(np.arr[arr[int]], *float, *float)
167             -> arr[arr[int(x2)](x4)]
168
169         '''
170         ymax, xmax = edges.shape[:2] # get height, width
171         # detect lines in the image
172         lines = cv2.HoughLinesP(edges, 1, np.pi/180, 100, minLineLength=50,
173                                 maxLineGap=50).reshape((-1,4))
174
175         # scale lines to the image boundaries
176         boundaries = [0,xmax,0,ymax]
177         for ind, line in enumerate(lines):
178             lines[ind] = Edge.scale_to_boundaries(line, boundaries)
```

```
179
180         # combine lines that are too similar (assumed to be the same line)
181         return Edge.reduce_lines(np.array(lines), col_thresh=col_thresh,
182                                  dist_thresh=dist_thresh)
183
184     @staticmethod
185     def _get_screen_points(lines):
186         ''' Returns the 4 corner points detected at 'lines' intersections.
187
188         If lines intersect in fewer than 4 places, returns None.
189         If lines intersect in more than 4 places, k-means clustering is used
190             to attempt to determine the 4 corner points.
191
192         Screen._get_screen_points(np.arr[arr[float(x4)]]) -> arr[arr[float(x2)]]
193
194         '''
195         ip = Edge.get_intersections(lines, positions=True)
196         # extract points from itersections' (ip[0]) positions (ip[1:3])
197         points = np.array(Edge.get_intersection_points(ip[1:3],ip[0]),
198                           dtype=np.float32)
199
200         # check if detected points are valid (>=4 --> 4, <4 --> invalid)
201         if len(points) > 4:
202             # use k-means clustering to find 4 most likely points
203             criteria = (0, 10, 1.0)
204             _, _, points=cv2.kmeans(points, 4, None, criteria, 10,
205                                     cv2.KMEANS_RANDOM_CENTERS)
206         elif len(points) < 4:
207             raise Exception('Insufficient points to detect screen')
208
209         # order points consistently
210         X, Y = np.array(points.T)
211         for i in range(4):
212             x,y = X[i], Y[i]
213             point = np.array([x,y])
214             if x < X.mean():
215                 if y < Y.mean():
216                     points[0] = point
217                 else:
218                     points[3] = point
219             else:
220                 if y < Y.mean():
221                     points[1] = point
222                 else:
223                     points[2] = point
224         return points
225
226     @staticmethod
```

```
227     def get_resolution(downsample=1):
228         ''' Returns the resolution of the screen — optionally downsampled. '''
229         return np.array(device.size()) // downsample
230 
231 
232 if __name__ == '__main__':
233     # test script for detecting a screen
234     import tkinter as tk
235     from time import sleep
236     width, height = device.size()
237     root = tk.Tk()
238     root.geometry('{}x{}'.format(width, height))
239     root.overrideredirect(True)  # allow true fullscreen
240     root.overrideredirect(False) # restore normal shortcuts
241     def fullscreen():
242         #root.config(bg='white')
243         root.update()
244         root.deiconify()
245         root.attributes('—fullscreen', True, '—topmost', True)
246         root.focus_force() # bring into focus
247     cap = cv2.VideoCapture(0)
248     if not cap.isOpened():
249         print('cannot open camera')
250         exit()
251     try:
252         while cv2.waitKey(1) != ord('c'):
253             success, frame = cap.read()
254             if success:
255                 cv2.imshow('frame', frame)
256             else:
257                 print('failure')
258         cv2.imwrite('proj.png', frame)
259         #while cv2.waitKey(1) != ord('c'): pass
260 
261         while cv2.waitKey(0) & 0xFF != ord('q'):
262             # capture frame—by—frame
263             fullscreen()
264             sleep(0.3)
265             read_success, frame = cap.read()
266             if not read_success:
267                 print("Can't receive frame (stream end?). Exiting...")
268                 break
269             root.withdraw()
270 
271             # operate on the frame
272             try:
273                 sc = Screen(Screen.points_from_img(frame, colour='W'))
274                 cv2.polylines(frame, np.int32([sc.points]),True,(0,0,255),2)
```

```
            except Exception:
                pass
            # display the result
            cv2.imshow('frame', frame)
    finally:
        # done, so release the capture
        cap.release()
        cv2.destroyAllWindows()
```

Appendix D

Edge Processing Code

Listing 4: Edge Processing Module

```
#!/usr/bin/env python3
##############################################
#                                            #
# Edge module, for lines and edges.          #
# Author: ES Alexander                       #
# Date: 6 Apr 2019                           #
# Last Modified: 14 Oct 2019                 #
#                                            #
##############################################
import numpy as np
from sys import float_info
epsilon = float_info.epsilon

class Edge(object):
    ''' A class for edge extraction and processing. '''
    @staticmethod
    def is_point(line):
        ''' Returns true if the specified line is in fact a point.

        'line' is a line segment of form [x0,y0,x1,y1].

        Edge.is_point(np.arr[float(x4)]) -> bool

        '''
        x0,y0,x1,y1 = line
        return x0 == x1 and y0 == y1

    @staticmethod
    def get_line_points(lines, *indices):
        ''' Returns a vector of points found in lines[indices] as [X,Y].

        Points are maintained in the order they appear in the relevant lines.

        'lines' is a vector of line segments in form [X0,Y0,X1,Y1].
```

```
        'indices' is a vector of indices with max(indices) < len(lines).

        Edge.get_line_points(np.arr[[float(x4)]], *int) -> arr[[float(x2)]]

        '''
        return lines[np.array(indices)].reshape(-1,2)

    @staticmethod
    def get_path_lines(path, closed=False):
        ''' Returns a vector of line segments traced by path, as [X0,Y0,X1,Y1].

        'path' is a vector of points in form [X0,Y0].
        'closed' is a boolean specifying if the path is closed - if True,
            creates a line segment from the last to the first point in 'path'.

        Edge.get_path_lines(np.arr[[float(x2)]], *bool) -> arr[[float(x4)]]

        '''
        lines = np.roll(path.repeat(2, axis=0), 1, axis=0).reshape((-1,4))
        if not closed:
            lines = lines[1:]
        return lines

    @classmethod
    def get_rect_lines(cls, rect):
        ''' Returns the lines that make up the rectangle 'rect'.

        'rect' is of the form [xmin, xmax, ymin, ymax].

        cls.get_rect_lines(np.arr[float(x4)])

        '''
        xmin, xmax, ymin, ymax = rect
        points = np.array([[xmin, ymin],[xmax,ymin],[xmax,ymax],[xmin,ymax]])
        return cls.get_path_lines(points, closed=True)

    @classmethod
    def get_path_interesection(cls, line, path_lines, position=False,
                               angle=False):
        ''' Returns True if line intersects with path_lines.

        'line' is of form [x0,y0,x1,y1].
        'path_lines' is of form [X0, Y0, X1, Y1].
        'position' is a boolean specifier determining if the position of a
            detected intersection point is returned.
        'angle' is a boolean specifier determining if the angle of a detected
            intersection between line and a path_line is returned.

```

```
        cls.get_path_interesection(np.arr[float(x4)], np.arr[[float(x4)]],
                                   *bool, *bool)

        '''
        for path_line in path_lines:
            output = cls.get_intersections(np.array([path_line, line]),
                                           dist_thresh=1,
                                           angles=angle, positions=position)
            intersection = output[0,0,1]
            if not intersection:
                continue
            if angle:
                i_angle = output[1,0,1]
                if position:
                    i_pos = output[2:,0,1]
                    return True, i_angle, i_pos
                return True, i_angle
            elif position:
                return True, output[1:,0,1]
            return True
        if angle and position:
            return False, None, None
        if angle or position:
            return False, None
        return False

    @staticmethod
    def get_path_length(path_points):
        ''' Returns the length of the path specified by path_points.

        'path_points' is an array of points, in form [[x0,x1,...,xn],...].

        Edge.get_path_length(np.arr[np.arr[float]]) -> float

        '''
        return np.sqrt((np.diff(path_points, axis=0) ** 2).sum(axis=1)).sum()

    @staticmethod
    def distsq(p0, p1):
        ''' Returns the squared Euclidian distance between p0 and p1.

        'p0' and 'p1' are points of form [x1,x2,...,xn], of same order.

        Edge.distsq(list[float], list[float]) -> float

        '''
        return sum((p1 - p0)**2)

```

```
    @classmethod
    def get_line(cls, points):
        ''' Returns a joined line segment covering the given points.

        Assumes the points are reasonably collinear and joins those furthest
            apart.

        'points' is an array of points in form [X,Y].

        cls.get_line(np.arr[[float(x2)]]) -> arr[float(x4)]

        '''
        num_points = len(points)
        max_dist = 0
        im = 0; jm = 0;
        for i in range(num_points):
            for j in range(i+1, num_points):
                new_dist = cls.distsq(points[i], points[j])
                if new_dist > max_dist:
                    max_dist = new_dist
                    im = i; jm = j
        return np.array([points[im], points[jm]]).reshape(4)

    @staticmethod
    def equal_lines(line0, line1):
        ''' Returns True if line0 and line1 are equivalent, else False.

        'line0' and 'line1' are line segments of form [x0,y0,x1,y1].

        Edge.equal_lines(np.arr[float(x4)], np.arr[float(x4)]) -> bool

        '''
        # break line segments into constituent points
        p0, p1 = line0.reshape((2,2))
        p0 = list(p0); p1 = list(p1)
        p2, p3 = line1.reshape((2,2))
        p2 = list(p2); p3 = list(p3)
        # compare in both directions
        return (p0 == p2 and p1 == p3) or (p0 == p3 and p1 == p2)

    @classmethod
    def add_new_lines(cls, lines, *extra_lines):
        ''' Returns lines with any additional extra_lines not already present.

        'lines' is a vector of line segments in form [X0,Y0,X1,Y1].
        'extra_lines' is an unspecified number of additional lines to add.

        cls.add_new_lines(np.arr[arr[float(x4)]], *arr[float(x4)])
```

```
            -> arr[arr[float(x4)]]

        '''
        new_lines = np.array(lines)
        for query_line in extra_lines:
            duplicate = False
            for existing_line in new_lines:
                if cls.equal_lines(existing_line, query_line):
                    duplicate = True
                    break
            if not duplicate:
                # genuine new line, add to list
                if new_lines.size > 0:
                    new_lines = np.r_[new_lines, [query_line]]
                else:
                    new_lines = np.array([query_line])
        return new_lines

    @staticmethod
    def duplicate_point(points):
        ''' Returns true if points contains multiples of one or more points.

        'points' is a list of points in form [X,Y].

        Edge.duplicate_point(list[list[float]]) -> bool

        '''
        return len(set(tuple(point) for point in points)) != len(points)

    @staticmethod
    def line_angle(line, degrees=True):
        ''' Returns the angle of 'line' with respect to the positive x-axis.

        Edge.line_angle(np.arr[float(x4)]) -> float

        '''
        x0,y0,x1,y1 = line
        angle = np.arctan2(y1-y0, x1-x0)
        if degrees:
            return np.degrees(angle)
        return angle

    @classmethod
    def angle_between_lines(cls, line0, line1):
        ''' Returns the angle (degrees) between the given lines.

        cls.angle_between_lines(np.arr[float(x4)], np.arr[float(x4)]) -> float

```

```
        '''
        return cls.line_angle(line1) - cls.line_angle(line0)

    @classmethod
    def dist_point_to_segment(cls, p0,p1,p2,positions=False):
        r''' Returns the closest distance from point p0 to segment p1-p2.

        'p0','p1','p2' are points of form [x,y].
        'positions' is a boolean specifier which, if True, appends the
            approximate point of intersection to the return value.

        Algorithm from:
        "https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line"

        cls.dist_point_to_segment(np.arr[float(x2)], arr[float(x2)],
                                  arr[float(x2)], *bool)
                -> float, *arr[float(x2)]

        '''
        d01sq = cls.distsq(p0,p1); d01 = np.sqrt(d01sq)
        d02sq = cls.distsq(p0,p2); d02 = np.sqrt(d02sq)
        d12sq = cls.distsq(p1,p2); d12 = np.sqrt(d12sq)

        if d01 == 0 or d02 == 0:
            # point on segment endpoint
            if positions:
                return 0, p0
            return 0
        elif d12 == 0:
            # segment is a point
            if positions:
                return d01, (p0 + p1)/2
            return d01
        u = np.round((d01sq + d12sq - d02sq) / (2 * d01 * d12), 10)
        if u <= 0:
            if positions:
                return d01, (p0 + p1)/2
            return d01
        elif d01 * u >= d12:
            if positions:
                return d02, (p0 + p2)/2
            return d02
        else:
            dist = d01 * np.sqrt(1 - u**2)
            if positions:
                pdiff = p2 - p1 + epsilon
                grad = pdiff[1]/pdiff[0]
                p4 = p1 + np.array([1,grad]) * dist / np.sqrt(grad**2 + 1)
```

```
                return dist, (p0 + p4)/2
            return dist

    @classmethod
    def min_dist_segments(cls, lines, position=False):
        ''' Returns the minimum distance  between the segments in 'lines'.

        'positions' is a boolean specifier which, if True, appends the
            approximate point of intersection to the return value.

        cls.min_dist_segments(np.arr[arr[float(x2)]], *bool)
                -> float, *arr[float(x2)]

        '''
        if position:
            output = cls.get_intersections(lines, positions=True)
            intersections = output[0]
            positions = output[1:]
        else:
            intersections = cls.get_intersections(lines)
        if np.abs(intersections).sum() == 2:
            # segments intersect
            if position:
                return 0, np.array([positions[0,0,1], positions[1,0,1]])
            return 0
        dist = np.Inf
        pos = None
        points = cls.get_line_points(lines, *range(len(lines)))
        point_triples = np.array([[0,2,3],[1,2,3],[2,0,1],[3,0,1]])
        for point in range(4):
            ps = points[point_triples[point]]
            if position:
                query_dist, query_pos = cls.dist_point_to_segment(*ps,
                        positions=True)
            else:
                query_dist = cls.dist_point_to_segment(*ps)

            if query_dist < dist:
                dist = query_dist
                if position:
                    pos = query_pos
        if position:
            return dist, pos
        return dist

    @staticmethod
    def offset_line(line, offset):
        ''' Returns 'line' offset by 'offset'.
```

```
323
324         'line' is a line segment of form [x0,y0,x1,y1].
325         'offset' is an [x,y] point which is added to 'line'.
326
327         Edge.offset_line(np.arr[float(x4)], arr[float(x2)]) -> arr[float(x4)]
328
329         '''
330         return (line.reshape((2,2)) + offset).reshape(4)
331
332     @staticmethod
333     def get_intersection_points(positions, intersections):
334         ''' Extracts and returns a list of [X,Y] points from 'positions'.
335
336         'positions' is an output array from Edge.get_intersections.
337
338         Edge.get_intersection_points(np.arr[arr[arr[float(xN)](xN)](x2)],
339                                      np.arr[arr[float(xN)](xN)])
340                 -> arr[arr[float(x2)](xN)]
341
342         '''
343         X = positions[0]
344         Y = positions[1]
345         n = len(X)
346         points = []
347
348         for i in range(n):
349             for j in range(i+1, n):
350                 if intersections[i,j]:
351                     points += [[X[i,j], Y[i,j]]]
352         return points
353
354     @staticmethod
355     def cross_orientation(a, b, c):
356         ''' Returns the orientation of the cross-product between points a,b,c.
357         Orientation is -1, 0, or +1.
358
359         'a','b','c' are [x,y] points, and have a 1 added for the cross product.
360
361         Edge.cross_orientation(np.arr[float(x2)], np.arr[float(x2)],
362                                np.arr[float(x2)]) -> int
363
364         '''
365         ax, ay = a; bx, by = b; cx, cy = c
366         return np.sign(np.linalg.det([[bx,by],[cx,cy]]) - \
367                        np.linalg.det([[ax,ay],[cx,cy]]) + \
368                        np.linalg.det([[ax,ay],[bx,by]]))
369
370     @classmethod
```

```
371     def line_intersection(cls, line0, line1):
372         ''' Returns True if line0 intersects line1, else False.
373         Returns -1 if line0 and line1 are collinear and intersect.
374 
375         cls.test_intersection(np.arr[float(x4)], np.arr[float(x4)]) -> int
376 
377         '''
378         p0, q0 = line0.reshape((2,2))
379         p1, q1 = line1.reshape((2,2))
380         ori_0p1 = cls.cross_orientation(p0, q0, p1)
381         ori_0q1 = cls.cross_orientation(p0, q0, q1)
382         ori_1p0 = cls.cross_orientation(p1, q1, p0)
383         ori_1q0 = cls.cross_orientation(p1, q1, q0)
384 
385         if ori_0p1 == ori_0q1 == 0 or ori_1p0 == ori_1q0 == 0:
386             return -1 # collinear
387 
388         return ori_0p1 != ori_0q1 and ori_1p0 != ori_1q0
389 
390     @staticmethod
391     def bounding_box_overlap(line0, line1):
392         ''' Returns True if the bounding boxes of line0 and line1 intersect.
393         Returns False otherwise, indicating a collision is impossible.
394 
395         Edge.bounding_box_overlap(np.arr[float(x4)], np.arr[float(x4)]) -> bool
396 
397         '''
398         x00, y00, x01, y01 = line0
399         x10, y10, x11, y11 = line1
400         x0_min = min(x00, x01)
401         x0_max = max(x00, x01)
402         x1_min = min(x10, x11)
403         x1_max = max(x10, x11)
404         if x0_max < x1_min or x1_max < x0_min:
405             return False
406 
407         y0_min = min(y00, y01)
408         y0_max = max(y00, y01)
409         y1_min = min(y10, y11)
410         y1_max = max(y10, y11)
411         if y0_max < y1_min or y1_max < y0_min:
412             return False
413 
414         return True
415 
416     @classmethod
417     def get_intersections(cls, lines, dist_thresh=0, angles=False,
418                           positions=False, i_min=0, j_min=0, i_max=None,
```

```
                    j_max=None):
        ''' Returns a len(lines)*len(lines)*([1—4]) array of intersections.

        Lines are considered not to be self—intersecting.

        If lines[i] intersects lines[j] | i!=j, intersections[i,j] = 1, unless
            the lines are collinear —> intersections[i,j] = —1.
            No intersection —> intersections[i,j] = 0.

        When positions are being returned, collinear intersections have their
            position as the center of the overlapping region.

        'lines' is a vector of line segments in form [X0,Y0,X1,Y1].
        'dist_thresh' is the allowable distance between segments within which
            an intersection is still registered.
        'angles' is a boolean specifier determining if intersection angles are
            included in the output matrix. (default False)
        'positions' is a boolean specifier determining if intersection positions
            are included in the output matrix. (default False)
        'i_min' specifies which line to start from. (default 0)
        'j_min' specifies which line to compare from. (default 0, but
            internally set to min(j_max, i+1) for each line i)
        'i_max' specifies which line to stop before. If left as None,
            checks intersections for all the lines after i_min.
        'j_max' specifies which line to stop comparing before. If left as None,
            checks intersections for all the lines after max(j_min, i+1).

        To determine intersections between lists lines_m and lines_n, use
            Edge.get_intersections(np.array(lines_m+lines_n), j_min=m, i_max=m)

        cls.get_intersections(np.array[np.arr[float]], *int, *bool, *bool,
                              *int, *int, *int, *int)
                -> np.arr[arr[float]],*arr[arr[float]],*arr[arr[float]]

        '''
        # initialise relevant state variables
        l = len(lines)
        intersections = np.zeros((l,l))
        kwargs = {}
        if angles:
            r_angles = np.zeros((l,l))
            kwargs['angles'] = r_angles
        if positions:
            r_positions = np.zeros((l,l,2))
            kwargs['positions'] = r_positions
        if i_max is None:
            i_max = l
        if j_max is None:
```

```
467             j_max = l
468
469         # iterate over the lines to check for intersection
470         for i in range(i_min, i_max):
471             for j in range(max(j_min, i+1), j_max):
472                 if not dist_thresh and \
473                    not cls.bounding_box_overlap(*lines[[i,j]]):
474                     continue # intersection not possible
475
476                 # extract a list of endpoints from the lines
477                 points = np.array(cls.get_line_points(lines, i, j))
478
479                 # check if the lines share an end-point
480                 if cls.duplicate_point(points):
481                     intersections[i,j] = True
482                     if angles:
483                         r_angles[i,j] = cls.angle_between_lines(*lines[[i,j]])
484                     if positions:
485                         for p0 in range(2):
486                             for p1 in range(2,4):
487                                 if (points[p0] - points[p1]).sum() == 0:
488                                     r_positions[i,j] = points[p0]
489                     continue # intersection handled
490
491                 # check if the lines intersect
492                 intersection = cls.line_intersection(*lines[[i,j]])
493                 if not dist_thresh:
494                     if not intersection:
495                         continue # lines don't intersect
496
497                 if positions:
498                     X, Y = points.T
499
500                 if intersection < 0:
501                     intersections[i,j] = -1 # collinear
502                     if angles:
503                         r_angles[i,j] = 0
504                     if positions:
505                         indices = np.ones(X.shape, dtype=bool)
506                         if cls.line_angle(lines[i]) in [0., -180., 180.]:
507                             var = X
508                         else:
509                             var = Y
510                         indices[[np.argmin(var), np.argmax(var)]] = False
511                         points = points[indices]
512                         r_positions[i,j] = points.mean(axis=0)
513                 elif intersection > 0:
514                     intersections[i,j] = 1 # standard intersection
```

```
                if angles:
                    r_angles[i,j] = cls.angle_between_lines(*lines[[i,j]])
                if positions:
                    # basis transform
                    # set min x point of line i as origin
                    x_min_index = np.argmin(X[:2])
                    offset = points[x_min_index]
                    offset_points = points - offset
                    # set line to 1st/4th quadrant vector for angle calc
                    line = list(offset) + list(points[(x_min_index+1)%2])
                    # rotate lines so line i is along +ve x-axis
                    theta = -cls.line_angle(line, degrees=False)
                    sin_t = np.sin(theta); cos_t = np.cos(theta)
                    rot_mat = np.array([[cos_t, -sin_t],
                                        [sin_t, cos_t]])
                    X, Y = np.dot(rot_mat, offset_points.T)
                    x_temp = X[2] - Y[2] * (X[3] - X[2]) / (Y[3] - Y[2])
                    rot_mat *= [[1,-1],[-1,1]] # reverse rotation angle
                    point = np.dot(rot_mat, [x_temp,0]).T
                    r_positions[i,j] = point + offset
            elif dist_thresh:
                # check for pseudo-intersection (if enabled)
                cls._pseudo_intersection(lines, i, j, intersections,
                                         dist_thresh, **kwargs)

    # convert from triangular to symmetric matrices for output
    #   diagonals all 0, so no need to halve
    intersections += intersections.T
    if angles:
        r_angles += r_angles.T
    if positions:
        r_positions += r_positions.transpose((1,0,2))
        # swap position axes to have x/y as first index, not row
        r_positions = r_positions.swapaxes(0,2).swapaxes(1,2)

    # generate output as selected by flags
    depth = 1 + angles + 2*positions
    output = np.zeros((depth,l,l))
    output[0] = intersections
    if angles:
        output[1] = r_angles
        if positions:
            output[2:] = r_positions
    elif positions:
        output[1:] = r_positions

    return output

```

```
    @classmethod
    def get_joined_lines(cls, lines, col_thresh, dist_thresh):
        ''' Returns the lines which can be joined with the given requirements.

        Also returns a list of truth values over the indices of the inputted
            lines, denoting whether or not they are involved in a line joining.

        'lines' is a vector of line segments in form [X1,Y1,X2,Y2].
        'col_thresh' is a collinearity threshold in degrees.
            Intersecting lines with an angle of less than col_thresh between
                them are joined together.
        'dist_thresh' is a distance threshold in coordinate units, allowing
            for pseudo-intersections to be detected if line-segments are within
            dist_thresh of one another.

        cls.get_joined_lines(list[list[int]], float, float)
                -> (list[list[int]], list[bool])

        '''
        joined_lines = []   # storage for joined lines
        l = len(lines)
        # track lines which have been joined to others
        joins = np.zeros(l, dtype=bool)

        intersections, angles = cls.get_intersections(lines, dist_thresh,
                                                      angles=True)

        # iterate over lines
        for i in range(l):
            for j in range(i+1,l):
                # if line i intersects with line j, and neither have already
                #   been joined to another line,
                if intersections[i,j] and not joins[i] and not joins[j]:
                    angle = angles[i,j]

                    # if lines are collinear within desired tolerance
                    if abs(angle) <= col_thresh or \
                       abs(180 - angle) <= col_thresh:
                        # lines are approximately collinear
                        #   -> (add both to joins list)
                        joins[i] = joins[j] = True
                        # get points of lines
                        points = cls.get_line_points(lines, i, j)
                        # determine endpoints of joined line
                        new_line = cls.get_line(points)
                        # add the new line to the list
                        joined_lines = cls.add_new_lines(joined_lines,
                                                         new_line)
```

```
611
612         return joined_lines, joins
613
614     @classmethod
615     def reduce_lines(cls, lines, col_thresh, min_changes=0, dist_thresh=0,
616                      stability_thresh=0):
617         ''' Returns the reduced lines based on the inputted minimum collinearity
618             and number of changes between iterations.
619
620         'lines' is a vector of line segments in form [X1,Y1,X2,Y2].
621         'col_thresh' is a collinearity threshold in degrees.
622             Intersecting lines with an angle of less than col_thresh between
623                 them are reduced to single lines
624         'min_changes' is an optimisation threshold.
625             If less than min_changes occur after an optimisation, the
626                 optimisation is stopped. (default 0)
627         'dist_thresh' is a distance threshold in coordinate units, allowing
628             for pseudo-intersections to be detected if line-segments are within
629             dist_thresh of one another. (default 0)
630         'stability_thresh' is a stability threshold for consecutive runs of
631             less than min_changes changes. (default 0)
632
633         cls.reduce_lines(np.arr[arr[float(x4)]], float, int, float, int)
634                 -> arr[arr[float(x4)]]
635
636         '''
637         # count the number of times min_changes has been reached
638         min_count = 0
639
640         while min_count <= stability_thresh:
641             l = len(lines)
642             # get all lines that can be joined together
643             new_lines, joins = cls.get_joined_lines(lines, col_thresh,
644                                                     dist_thresh)
645             # add unjoined lines to new_lines (they're still valid)
646             new_lines = cls.add_new_lines(new_lines, *lines[np.invert(joins)])
647             num_changes = abs(l - len(new_lines))
648             lines = new_lines
649             if num_changes <= min_changes:
650                 min_count += 1
651             else:
652                 min_count = 0
653
654         return new_lines
655
656     @staticmethod
657     def is_out_of_bounds(query_point, boundaries):
658         ''' Returns True if the query point is outside the boundaries,
```

```
659             else False.
660
661         'queryPoint' is an [x,y] coordinate
662         'boundaries' is a list of [xmin, xmax, ymin, ymax]
663             -> NOTE: boundaries can be +/-Inf if only some are desired
664
665         Edge.is_out_of_bounds(np.arr[float(x4)], arr[float(x2)]) -> bool
666
667         '''
668         x_min, x_max, y_min, y_max = boundaries
669         x_q, y_q = query_point
670         return not (x_min <= x_q <= x_max and y_min <= y_q <= y_max)
671
672     @classmethod
673     def scale_lines(cls, lines, scale_factor, boundaries=[-np.Inf,np.Inf]*2):
674         ''' Returns 'lines' scaled about their centres by 'scale_factor'.
675
676         'lines' is a vector of line segments in form [X1,Y1,X2,Y2].
677         'scaleFactor' is a +ve float, 1 is unity scale.
678         'boundaries' is a list of [xmin, xmax, ymin, ymax], which, if provided,
679             limit the scaling range of each line. Lines scaled with boundaries
680             in place are extended to the minimum of the new extension and the
681             boundary. Boundaries can be None if only some are desired.
682
683         Algorithm ignores the possibility of a line detected that lies on a
684             boundary line.
685
686         cls.scale_lines(np.arr[arr[float(x4)]], float, *list[float(x4)])
687                 -> arr[arr[float(x4)]]
688
689         '''
690         lines = np.array(lines)
691         new_lines = lines # initialise to old lines
692         # logical array of line validity - false excluded at the end
693         include = np.ones(len(lines), dtype=bool)
694
695         # generate boundary lines
696         xmin, xmax, ymin, ymax = boundaries
697         boundary_lines = np.array(
698             [[xmin, ymin, xmin, ymax], #xmin
699              [xmax, ymin, xmax, ymax], #xmax
700              [xmin, ymin, xmax, ymin], #ymin
701              [xmin, ymax, xmax, ymax], #ymax
702              np.zeros(4)])
703
704         # iterate over lines
705         for line_ind in range(len(lines)):
706             line = lines[line_ind]
```

```
            # check if line is a point (remove from list if so)
            if cls.is_point(line):
                include[line_ind] = False
                continue

            if scale_factor != 1:
                # get current points
                p0, p1 = line.reshape((2,2))
                midpoint = (p0 + p1) / 2;
                # scale points about midpoint
                p_new = np.array(
                    [(p0 - midpoint) * scale_factor + midpoint,
                     (p1 - midpoint) * scale_factor + midpoint])
            else:
                p_new = line.reshape((2,2))

            # check boundary conditions, fix if required
            p0out = cls.is_out_of_bounds(p_new[0], boundaries)
            p1out = cls.is_out_of_bounds(p_new[1], boundaries)

            # check if scaled points are outside the boundary
            if p0out or p1out:
                # add new line to boundary lines list
                boundary_lines[4] = p_new.reshape(4)

                # get boundary intersections
                ip = cls.get_intersections(boundary_lines, positions=True)
                intersects = np.array(np.abs(ip[0,4]), dtype=bool)
                num_intersections = intersects.sum()
                new_poss = np.unique(np.array([ip[1,4], ip[2,4]]).T[intersects],
                                     axis=0)
                num_crosses = len(new_poss)

                if num_intersections == 0:
                    # line is fully outside the boundaries (remove)
                    include[line_ind] = False
                    continue
                if num_crosses == 1:
                    # 1 out, 1 in (redefine one point)
                    new_point = new_poss
                    if p0out:
                        p_new[0] = new_point
                    else:
                        p_new[1] = new_point
                elif num_crosses == 2:
                    # 2 boundaries uniquely intersected with, line redefined by
                    #   its boundary intersections
                    p_new = new_poss
```

```
                else:
                    # NOT REACHED (or at least it shouldn't be...)
                    print('NO!', num_intersections, num_crosses, ip, new_poss)
            # replace old points with the new ones
            new_lines[line_ind] = p_new.reshape(4)
        # remove out of bounds lines
        return new_lines[include]

    @staticmethod
    def scale_to_boundaries(line, boundaries):
        ''' Returns 'line' scaled to the provided boundaries.

        'line' is of the form [x0,y0,x1,y1].
        'boundaries' is of the form [xmin, xmax, ymin, ymax].

        Edge.scale_to_boundaries(np.arr[float(x4)], arr[float(x4)])
                -> np.arr[float(x4)]

        '''
        x0,y0,x1,y1 = line
        x0p, y0p, x1p, y1p = x0, y0, x1, y1 # copy for use in functions
        y = lambda x : (y1p - y0p) * (x - x0p) / (x1p - x0p + 1e-9) + y0p
        x = lambda y : (x1p - x0p) * (y - y0p) / (y1p - y0p + 1e-9) + x0p
        xmin, xmax, ymin, ymax = boundaries
        # scale to x boundaries
        x0, y0 = xmin, y(xmin)
        x1, y1 = xmax, y(xmax)
        # if outside y boundaries, scale back to y-boundaries
        if y0 < ymin or y0 > ymax:
            y0 = ymin if y0 < ymin else ymax
            x0 = x(y0)
        if y1 < ymin or y1 > ymax:
            y1 = ymin if y1 < ymin else ymax
            x1 = x(y1)
        return np.array([x0,y0,x1,y1], dtype=int)

    @classmethod
    def _pseudo_intersection(cls, lines, i, j, intersections, dist_thresh,
                             **kwargs):
        ''' Evaluates if a pseudo-intersection has occurred.

        Updates relevant variables if an intersection is registered.

        cls._pseudo_intersection(list[list[float]], int, int, float,
                                 list[list[bool]]) -> None

        '''
        # parse kwargs
```

```
803         angles = kwargs.pop('angles', None)
804         calculate_angle = angles is not None and len(angles) > 0
805         positions = kwargs.pop('positions', None)
806         calculate_position = positions is not None and len(positions) > 0
807 
808         # determine if pseudo-intersection has occurred.
809         dist = cls.min_dist_segments(lines[[i,j]], position=calculate_position)
810         if calculate_position:
811             dist, pos = dist # extract correct variables from result
812 
813         if dist < dist_thresh:
814             intersections[i,j] = True
815             if calculate_angle:
816                 angles[i,j] = cls.angle_between_lines(lines[i], lines[j])
817             if calculate_position:
818                 positions[i,j] = pos
```

Appendix E

Interaction Detection Code

Listing 5: Interaction Detector - Abstract Code

```
#!/usr/bin/env python3

import numpy as np
import cv2
from abc import ABC as AbstractClass
from abc import abstractmethod
from time import time # for Kalman filtering

class InteractionDetector(AbstractClass):
    r''' An abstract interface for detecting interactions in an image.

    The Kalman filtering included in this class contains code adapted in part
    from 'https://www.myzhar.com/blog/tutorials/tutorial-opencv-ball-tracker-
    using-kalman-filter/', available under LGPLv3.0. It has been modified to
    track just position and velocity, changed from C++ to Python, and does not
    use the ball-tracker image processing.

    '''
    def __init__(self, *kalman_args, kalman=True, not_found_thresh=2,
                 **kalman_kwargs):
        ''' Creates an interaction detector instance.

        Kalman compensation (if used) assumes tracking a single point over x,y
        space, with velocity.

        'kalman' is a boolean specifier determining if Kalman compensation is
            used to predict the state of the interaction on demand, corrected
            for with measurements.
        'kalman_args' and 'kalman_kwargs' are arguments and keyword-arguments
            passed to cv2.KalmanFilter.
        'not_found_thresh' is an integer limit on how many subsequent
            predictions are allowed once detection has dropped out, generally
            to attempt to continue tracking in temporary losses of the
            interaction, due to occlusions and the like.
```

```
35
36          '''
37          if kalman:
38              self.init_kalman(*kalman_args, **kalman_kwargs)
39          self._found = False
40          self._not_found_count = 0
41          self._not_found_thresh = not_found_thresh
42          self._kalman = kalman
43
44      def init_kalman(self, state_size=4, measurement_size=2, *kalman_args,
45                      **kalman_kwargs):
46          ''' Creates and initialises a Kalman filter to predict the state of
47              the interaction with consideration of its position and velocity.
48
49          Used as compensation for delay, to track the interaction more closely
50              than pure measurements.
51
52          Arguments are passed to a cv2.KalmanFilter instance.
53
54          '''
55          kf = cv2.KalmanFilter(state_size, measurement_size, *kalman_args,
56                                **kalman_kwargs)
57          # Transition State Matrix (set dT at each processing step)
58          # [ 1 0 dT 0  ]
59          # [ 0 1 0  dT ]
60          # [ 0 0 1  0  ]
61          # [ 0 0 0  1  ]
62          kf.transitionMatrix = np.eye(state_size, dtype=np.float32)
63
64          # Measurement Matrix
65          # [ 1 0 0 0 ]
66          # [ 0 1 0 0 ]
67          kf.measurementMatrix = np.zeros((measurement_size, state_size),
68                                          dtype=np.float32)
69          kf.measurementMatrix[[0,1],[0,1]] = np.float32(1.0)
70
71          # Process Noise Covariance Matrix
72          # position and speed error assumed uncorrelated
73          # [ Ex 0  0    0    ] pixels
74          # [ 0  Ey 0    0    ] pixels
75          # [ 0  0  Ev_x 0    ] pixels/second
76          # [ 0  0  0    Ev_y ] pixels/second
77          kf.processNoiseCov = np.eye(state_size, dtype=np.float32)
78          if state_size == 2 * measurement_size:
79              # assume first half are position, second half velocity
80              processNoise = [500]*measurement_size + [10]*measurement_size
81              for i, E in enumerate(np.float32(processNoise)):
82                  kf.processNoiseCov[i,i] = E
```

```
        else:
            kf.processNoiseCov *= np.float32(1e-2)

        # Measurement Noise Covariance Matrix
        kf.measurementNoiseCov = np.eye(measurement_size,
                                        dtype=np.float32) * 1e-2 # pixel

        self._kf = kf
        self._correction_time = time()
        self._state_size = state_size

    @abstractmethod
    def detect_interaction(self, img):
        ''' Attempts to detect an interaction in the image, updating the
            internal state.

        To get the latest estimate for the interaction location, use
            self.predict().

        'img' is the image being processed.

        self.detect_interaction(np.arr[arr[int](x3)]) -> None

        '''
        pass

    def _update_time(self):
        ''' Updates transition matrix with the time since the last correction.

        Not used when Kalman filter compensation is off.

        By default assumes position and velocity tracking of a single point in
            2D space.

        '''
        self._kf.transitionMatrix[[0,1],[2,3]] = \
                np.float32(time() - self._correction_time)

    def _handle_detection(self, detected, measurement):
        ''' Handles a detection attempt, tracking if the target is found, and
            updating the kalman filter with valid measurements (if used).

        'detected' is a boolean specifying if the interaction was detected in
            the attempt being handled.
        'measurement' is the detected measurement, or not used.

        self._handle_detection(bool, np.array[float32]) -> None

```

```
        '''
        if not detected and self._found:
            self._not_found_count += 1
            if self._not_found_count > self._not_found_thresh:
                self._found = False
        elif detected:
            self._not_found_count = 0
            self._last = measurement
            if not self._found: # first detection
                self._found = True
                if self._kalman:
                    self._kf.errorCovPre = np.eye(self._state_size,
                                                  dtype=float) * 1.0 # px
                    self._kf.statePost = np.array([*measurement, 0, 0],
                                                  dtype=np.float32)
                    self._correction_time = time()
            elif self._kalman:
                self._correct(np.array(measurement, dtype=np.float32))
        # else not tracking

    def predict(self):
        ''' Return the currently expected state from the filter.
        If not currently found/tracking, returns None.
        If not using Kalman filtering, returns last detected measurement.

        self.predict() -> arr[int(x2)]/None

        '''
        if not self._found:
            return None

        if not self._kalman:
            return np.array(self._last)

        self._update_time()
        return self._kf.predict()

    def _correct(self, measurement):
        ''' Corrects the state of the Kalman Filter with measurement.

        'measurement' should be an array of 32-bit floats.

        '''
        self._update_time()            # update to the current point in time
        self._kf.correct(measurement)  # correct with the new measurement
        self._correction_time = time() # set now as the latest correction time
```

Listing 6: Laser Detector Implementation

```
#!/usr/bin/env python3

from interaction_detector import * # np, cv2, InteractionDetector
from overrides import overrides

class LaserDetector(InteractionDetector):
    ''' A class for detecting laser-pointer interactions with a screen. '''
    def __init__(self, colour='g', state_size=4, measurement_size=2,
                 *kalman_args, kalman=True, **kalman_kwargs):
        ''' Creates an interaction detector for tracking a laser-pointer.

        'colour' is the colour of the laser-pointer being tracked.
            Options are 'g'/'green', 'r'/'red', 'p'/'purple', and 'b'/'blue'.

        The remaining arguments are passed to a cv2.KalmanFilter instance
            to compensate for processing delay between measurements, unless
            'kalman' is set to False, in which case no compensation is used.

        '''
        super().__init__(state_size, measurement_size, *kalman_args,
                         **kalman_kwargs)

        self.colour = colour.lower()
        # create a kernel to help look for a laser-pointer within an image.
        self._kernel = np.array([[-4,-4,-4,-4,-4],
                                 [-4,-3, 2,-3,-4],
                                 [-4, 2,68, 2,-4],
                                 [-4,-3, 2,-3,-4],
                                 [-4,-4,-4,-4,-4]])
        self._first = True

    @overrides
    def detect_interaction(self, img):
        if len(img.shape) == 2 or img.shape[2] == 1:
            grey = img
        else:
            # convert to float image, and split into colour channels
            b,g,r = cv2.split(img / 255.0)

            if self.colour in ['g', 'green']:
                grey = g
            elif self.colour in ['r', 'red']:
                grey = r
            elif self.colour in ['p', 'purple']:
                grey = (r + b) / 2
            elif self.colour in ['b', 'blue']:
```

```
            grey = b
        else:
            # colour not supported
            grey = cv2.COLOUR_BGR2GRAY(img)

    # attempt to detect laser-pointer shaped objects
    filtered = cv2.filter2D(np.array(grey, dtype=np.float32), -1,
                            self._kernel)
    # filter out previous to remove DC errors
    if not self._first:
        adjusted = filtered - self._prev
        adjusted[adjusted < 0] = 0
    self._prev = filtered
    if self._first:
        self._first = False
        return
    blur = cv2.GaussianBlur(adjusted, (9,9), 0)

    # find the strongest point of detection in the image
    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(blur)
    if maxVal > 0.3:
        detection = True
        measurement = maxLoc
    else:
        detection = False
        measurement = None
    self._handle_detection(detection, measurement)
    return
```

Appendix F

Data Analysis Code

Listing 7: Code for segmenting and summarising attempts from testing

```
#!/usr/bin/env python3

import numpy as np
from Edge2 import Edge
from os import path

class Attempt(object):
    ''' A class for storing a maze attempt. '''
    def __init__(self, points, dTs, lines, classes, maze_lines):
        ''' Construct a maze attempt using the collected data. '''
        self._points = points
        self._dTs = dTs
        self._lines = lines
        self._classes = classes
        self._maze_lines = maze_lines
        self._stats = None

    def analyse(self):
        ''' Analyse the internal data and update internal statistics. '''
        stats = {'t90':0, 't180':0, 't0':0, 'outside':0, 'total_time':0,
                 'failures':0}

        l = len(self._lines)
        intersections = np.abs(Edge.get_intersections(
                np.array(self._lines+self._maze_lines), j_min=l, i_max=l))

        for index, dT in enumerate(self._dTs):
            # can't account for intersections because don't know if return
            #   to correct region before continuing
            stats[self._classes[index]] += dT
            stats['total_time'] += dT

        stats['failures'] = intersections.sum() / 2
        stats['distance'] = Edge.get_path_length(self._points)
```

```
35
36          self._stats = stats
37
38      def get_stats(self, force_recompute=False):
39          ''' Retrieve the dictionary of statistics for this attempt. '''
40          if self._stats is None or force_recompute:
41              self.analyse()
42          return self._stats
43
44  class Analyser(object):
45      ''' A class for detecting and analysing maze attempts in a test run. '''
46      def __init__(self, points, maze_regions, maze_lines):
47          ''' Construct the analyser and perform basic pre-processing. '''
48          X, Y, t = np.array(points).T
49          self._points = np.array([X,Y]).T
50          self._maze_regions = maze_regions
51          self._maze_lines = maze_lines
52          self._classify_points()
53          self._dTs = [*np.diff(t)]
54          self._lines = Edge.get_path_lines(self._points).tolist()
55          self._attempts = []
56          self._split_attempts()
57          self._stats = None
58
59      def _classify_points(self):
60          ''' Classifies points based on the maze region they exist within. '''
61          self._point_classes = []
62          for index, point in enumerate(self._points):
63              classified = False
64              for region in self._maze_regions:
65                  if not Edge.is_out_of_bounds(point, self._maze_regions[region]):
66                      # point is in this region
67                      self._point_classes.append(region.split('_')[0])
68                      classified = True
69                      break
70              if not classified:
71                  self._point_classes.append('outside') # point not in maze
72
73      def _split_attempts(self):
74          ''' Split the stored data into classes.
75
76          Can be performed multiple times, but only works for attempt data added
77              since the last split, not if previously split data is modified.
78
79          '''
80          started = False
81          ended = True
82          for index, region in enumerate(self._point_classes):
```

```
            if not started and region == 'start':
                start = index
                started = True
                ended = False
            elif started and not ended and region == 'end':
                ended = True
            elif started and ended and region != 'end':
                self._attempts.append(Attempt(self._points[start:index],
                                              self._dTs[start:index],
                                              self._lines[start:index],
                                              self._point_classes[start:index],
                                              self._maze_lines))
                started = False
            if region in ['start', 'end']:
                self._point_classes[index] = 't90'
        print('recorded {} attempts'.format(len(self._attempts)))

    def analyse(self, force_recompute=False):
        ''' Analyse the stored attempts and store results internally. '''
        stats = []
        for attempt in self._attempts:
            stats.append(attempt.get_stats(force_recompute))
        self._stats = stats

    def get_stats(self, force_recompute=False):
        ''' Returns the list of attempt analysis statistics (dicts). '''
        if self._stats is None or force_recompute:
            self.analyse(force_recompute)
        return self._stats

    def save_stats(self, filename='stats'):
        ''' Save the internal statistics to the given filename +.csv. '''
        suffix=0
        while path.exists(filename+str(suffix)+'.csv'):
            suffix += 1
        filename = filename + str(suffix) + '.csv'

        with open(filename, 'w') as out:
            first = True
            for attempt_stats in self.get_stats():
                if first:
                    out.write(','.join(attempt_stats.keys()) + '\n')
                    first=False
                out.write(','.join(str(v) for v in attempt_stats.values()) \
                          + '\n')
        return filename
```