Linköping University | Department of Computer Science
Bachelor thesis, 16 ECTS | Datateknik
2016 | LIU-IDA/LITH-EX-G--16/036--SE

Lightweight User Agents

Användaragenter med små avtryck

Martin Estgren

Supervisor : Teodor Sommestad
Examiner : Cyrille Berger

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

The unit for information security and IT architecture at The Swedish Defence Research Agency (FOI) conducts work with a cyber range called CRATE (Cyber Range and Training Environment). Currently, simulation of user activity involves scripts inside the simulated network. This solution is not ideal because of the traces it leaves in the system and the general lack of standardised GUI API between different operating systems. FOI are interested in testing the use of artificial user agent located outside the virtual environment using computer vision and the virtualisation API to execute actions and extract information from the system.

This paper focuses on analysing the reliability of template matching, a computer vision algorithm used to localise objects in images using already identified images of said object as templates. The analysis will evaluate both the reliability of localising objects and the algorithms ability to correctly identify if an object is present in the virtual environment.

Analysis of template matching is performed by first creating a prototype of the agent's sensory system and then simulate scenarios which the agent might encounter. By simulating the environment, testing parameters can be manipulated and monitored in a reliable way. The parameters manipulated involves both the amount and type of image noise in the template and screenshot, the agents' discrimination threshold for what constitutes a positive match, and information about the template such as template generality.

This paper presents the performance and reliability of the agent in regards to what type of image noise affects the result, the amount of correctly identified objects given different discrimination thresholds, and computational time of template matching when different image filters are applied. Furthermore the best cases for each study are presented as comparison for the other results.

In the end of the thesis we present how for screenshots with objects very similar to the templates used by the agent, template matching can result in a high degree of accuracy in both object localization and object identification and that a small reduction of similarity between template and screenshot to reduce the agent's ability to reliably identifying specific objects in the environment.

Acknowledgments

I would like to thank FOI and especially my supervisor Teodor Sommestad for giving me the opportunity to do my bachelor thesis with them and for doing a great job at aiding me during the course of the project. I've learned a lot during the last couple of months.

Furthermore, I would like to thank my examiner Cyrille Berger for his guidance during this project.

At last I would like to thank Rasmus Holm and Adnan Avdagic for both providing feedback on the report and aiding me by discussing different approaches and solutions during the project.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The unit for information security and IT architecture at The Swedish Defence Research Agency (FOI) conducts work with a cyber range, a virtual computer environment used for cyberwarfare training, called CRATE [1] (Cyber Range And Training Environment). Cyberwarfare can be defined in a multitude of ways [1] but common for all of them is that they involve one or more organisations trying to disrupt and/or damage another party's digital communication. An example of such action can be Distributed Denial of service attacks (DDOS) like the one hitting Github in 2015 [2].

CRATE consists of a server cluster about 800 units in size and utilises the virtualization utility called VirtualBox to simulate different kinds of systems [1]. Network units that cannot be simulated in VirtualBox are connected either physically or through a virtual private network (VPN).

In the case of many cyber ranges it is desirable create an as authentic as possible virtual environment. The reason for this is that the more authentic the environment, the more the data gathered can be generalised to other live systems. To achieve as high level of authenticity, user activity have to be simulated in the network. This can for example be realised either by using scripts located inside the virtual environment or using so called artificial users that read emails, surf the web, and install software.

In the case of CRATE the user activity is simulated using scripts placed within the network. A more desirable solution would involve the user agents running outside the virtual environment and interact with the system thought the virtualization API by using virtual hardware adapters for peripherals such as mouse and keyboards. In order for the agent to plan and evaluate what actions to perform, information about the environment is needed. By using computer vision to analyse screenshots taken from the environment the agent can figure out what programs that are currently running, providing a information basis for its decisions.

OpenCV is a software library that contains software used for computer vision [2]. The library contains algorithms used for face recognition, object tracking, image analysis tools,

[1]`https://en.wikipedia.org/wiki/Cyberwarfare`

[2]`https://github.com/blog/1981-large-scale-ddos-attack-on-github-com`

etc. In this thesis template matching implemented in OpenCV will be examined to determine if it is a suitable technique for the agent to use as sensors.

1.2 Aim

The study's main goal is to provide analysis of the suitability and reliability of template matching used in GUI environment for localization and identification of objects. For this purpose a prototype agent will be developed. The agent shall be able to extract information about the graphical user interface from screenshots using template matching.

1.3 Research questions

1. Can template matching be used to reliably identify elements in a graphical user interface?
2. Can a virtual user agent use this data in order to reliably navigate the environment?
3. What is the performance of the developed agent in terms of computational time?

1.4 Delimitations

The final purpose of the agent architecture is to be able to run a large (potentially thousands) amount of them simultaneously in order to simulate user activity a large scale virtual network. This study is on the other hand aimed at evaluate the reliability of template matching and the performance of the agent in terms of computational time and will thereby not focus on questions about running multiple instances concurrently.

2 Theory

The theoretical background for study is split into the following sections: digital image processing, intelligent agent design and the statistical analysis methods used in this thesis.

2.1 Digital Image Processing

Feature Detection

A *Feature* is a part of an image or a pattern that has a certain characteristic making it identifiable. This can be a complete object such as a ball or part of an object such as the corner of a building. *Feature matching* is a technique that can be used for object identification and tracking across multiple images [2].

Common techniques used for *feature matching* involves transforming interesting regions of an image to a structure that the computer can understand and compare [2]. Some of the common approaches involves identifying the following aspects of an image:

- Corners
- Edges
- Blobs

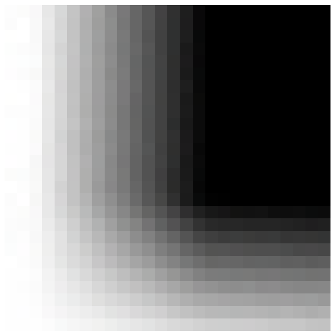


Figure 2.1: Example of an image corner.

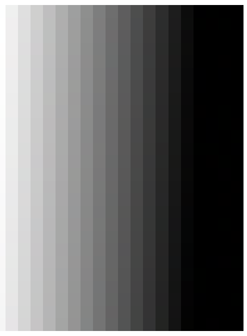


Figure 2.2: Example of an image edge.

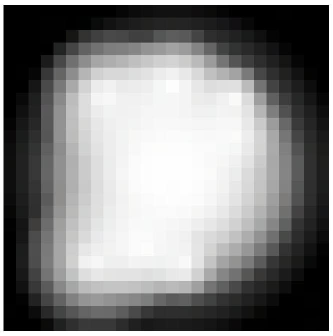


Figure 2.3: Example of an image blob/region.

In digital image processing a corner is defined as a point in an image where there is a high degree of intensity variance in at least two directions. Example of an image corner can be viewed in figure.2.1. Edges requires only a high degree of intensity variance in a single direction as can be observed in figure.2.2. Blobs or regions are areas in an image isolated by corners and edges. An example of a blob can be observed in the example figure.2.3.

Scale Invariant Feature Transform

Scale Invariant Feature Transform (SIFT) is used to detect corners in images that are independent of affine transformation such as translation, scaling, rotation, and illumination. The algorithm was created by David. G. Lowe in 1999 [3]. These properties makes the algorithm attractive for identifying and tracking objects in a 3D environment where the perspective of an object is changing. The algorithm is split up into the following steps.

- Create a *scale space* from the image.
- Apply *Difference of Gaussians* to the *scale space*.
- Find key points.
- Remove low-contrast key points and edge-points.
- Index the remaining key points.
- Generate *(*features) from the key points.

The first step, generating a *scale space* involves generating progressively blurred and scaled images of the source image using *Gaussian Blur*.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.1}$$

Where $L(x, y, \sigma)$ is the blurred image, x, y is the point in the image, σ specifies the amount of blur to apply and $G(x, y, \sigma)$ is the Gaussian Blur function:

$$\frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{2.2}$$

The Gaussian function is convoluted over the source image $I(x, y)$. This procedure is repeated for each scale level.

The next step involves applying a *Difference of Gaussians* function to the scale space in order to find scale-invariant key points. The *Difference of Gaussians* is defined as identical images with different strengths of *Gaussian Blurs* applied, subtracted from each other, leaving only the edges in the image.

$$G(x,y,\sigma_1) - G(x,y,\sigma_2) = \frac{1}{\sqrt{2\pi}}(\frac{1}{\sigma_1}e^{-(x^2+y^2)/2\sigma_1^2} - \frac{1}{\sigma_2}e^{-(x^2+y^2)/2\sigma_2^2}) \tag{2.3}$$

The local extremium in the resulting image are considered key points. For each key-point candidate, the local extremium is approximated using *Taylor Expansion* applied on the area around the targeted key point.

$$D(x) = D + \frac{\partial D^T}{\partial x}x + \frac{1}{2}x^T\frac{\partial^2 D}{\partial x^2}x \tag{2.4}$$

where $x = (x,y,\sigma)^T$ is the offset from the key point. The suitability of the approximated extremium $\hat{x}$ is determined by taking the derivative of the function 2.4 with regards to x and look for points where $D'(x) = 0$ resulting in $\hat{x} = -\frac{\partial^2 D^{-1}}{\partial x^2}\frac{\partial D}{\partial x}$. If the offset $\hat{x}$ is larger than 0.5 in any direction, the extreme point lies closer to another candidate point and the approximation is performed for that point instead. If the offset $\hat{x}$ is lower than 0.03 it is discarded as a low contrast key point.

The next step involves removing all edge-points. This is done because edge-points are not robust enough features to be included in the matching procedure. Algorithmically this is performed by calculating the eigenvalues for the second-order *Hessian Matrix*:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{2.5}$$

In practice only the ratio $r = \frac{\alpha}{\beta}$ of the eigenvalues α and β where $\alpha > \beta$ is precise enough for this purpose. The ratio:

$$R = \frac{(D_{xx} + D_{yy})^2}{D_{xx}D_{yy} - D_{xy}^2} \tag{2.6}$$

can because of its equivalence with $(r+1)^2/r$ be used to calculate the ratio of the eigenvalues without first calculating the values respectively. If r is larger than $(r_{th}+1)^2/r_{th}$ given a threshold r_{th} the key point is kept, otherwise rejected.

After this step only the key points determined robust enough are left. The next step in *SIFT* focuses on assigning an orientation to each key point. This will make the key points rotation invariant. The orientation is decided by analysing the intensity-gradient around each key point according to the following function:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{2.7}$$

$$\theta(x,y) = atan2(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}) \tag{2.8}$$

where $m(x,y)$ is the magnitude and $\theta(x,y)$ is the direction of the gradient. A histogram is created with each bin representing 10 degrees, resulting in 36 bins in total. Each neighbouring sample around a key point is added to the histogram and weighted by its magnitude and a Gaussian circular window 2.4 where the weights are reduced the further from the middle, with a scale of 1.5 that of the key point. Once the histogram is filled, values at least 80% of the peak value are selected. These are the dominant orientations of the key point. For each of these values a key point with the same scale as the original is assigned to the value. This is done because each key point cannot have more than one dominant orientation.

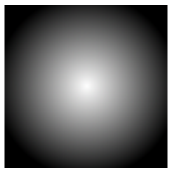


Figure 2.4: Example of a *Gaussian Window* with the size of 128.

The final step in the algorithm involves creating feature descriptors. The descriptors should be as invariant to luminance shift and perspective change as possible. First a 16x16 window around the key point is defined and subdivided into 4x4 windows. Within each of these 4x4 windows the magnitude and orientation gradients are calculated and put into a histogram of 8 bins with each bin representing 45 degrees. Each point added to the histogram is weighted with a Gaussian function with $\sigma = 1/2$ width of the window. All values in the histogram get turned into a vector and then normalised. The resulting vector is very distinct and can be used to reliably identify the same or similar key points in different images.

Harris Corner Detection

Harris Corner Detection[4] as described by Harris and Stephens as an algorithm used for finding corners in images. The algorithm builds on the *Moravec corner detector algorithm* [5] and consists of the following steps:

First the intensity gradient in all directions from a given point (u, v) is calculated. To do this the following function is defined as the following set of equations:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,.y+v) - I(x,y)]^2 \tag{2.9}$$

where $w(x,y)$ is the weighted window function using a Gaussian window and $[I(x+u, y+v) - I(x,y)]^2$ is the *squared difference* in intensity between two points.

For corner detection the $E(u,v)$ should be large as possible and for this *Tailor Expansion* is performed with I_u and I_v as partial derivatives of I:

$$I(x+u, y+v) \approx I(x,y) + I_u(x,y)u + I_v(x,y)v \tag{2.10}$$

producing the approximation:

$$E(u,v) = \sum_{x,y} w(x,y)[I_u(x,y)u + I_v(x,y)v]^2 \tag{2.11}$$

which can be written in the matrix form

$$E(u,v) \approx [u,v]M \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.12}$$

where M is equal to:

$$\sum_{x,y} w(u,v) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix} \tag{2.13}$$

I_x and I_y are the intensity gradient in x and y directions. Now let λ_1 and λ_2 be eigenvalues of M and the scoring function, used for determining if an window contains a corner be:

$$R = det(M) - k(trace(M))^2 \tag{2.14}$$

where $det(M) = \lambda_1 \lambda_2$ and $trace(M) = \lambda_1 + \lambda_2$. If both λ_1 and λ_2 both are large and $\lambda_1 \sim \lambda_2$ the window contains a corner.

Feature matching

Brute-force matcher

Brute-force matcher [6] performs feature matching for each feature from one image against all features extracted from a second image. The distance is calculated using a distance metric such as *euclidean distance* $|x| = \sqrt{\sum_{k=1}^{n} x_k^2}$ or *hamming distance* $\sum_{i=1}^{k} (x_i - y_i)$ where $x, y \in \{0, 1\}$ for binary feature descriptors.

Fast Library for Approximate Nearest Neighbours

FLANN [7] or Fast Library for Approximate Nearest Neighbour is a library used in OpenCV to perform feature matching using the *approximate nearest neighbour* function with the k-d tree data structure. FLANN can use the same distance metrics as *brute-force matching* but also incorporates an optimised data structure to minimise the search time. FLANN is commonly used when large sets of features needs to be searched and brute-force matching isn't an option.

Template Matching

For situations where looking for static objects that does not significantly change appearance in between images *Template Matching* can be used [8, 2]. The way it works is by sliding an image (template) of an object over a large image that might contain said object. For each position of the template a value representing the similarity between the template and the underlying region of the larger image is recorded. The similarity-measures implemented in OpenCV are the following:

- Cross Correlation
- Squared Difference
- Cross coefficient correlation

The next sections will explain the different similarity measures in a more elaborate way.

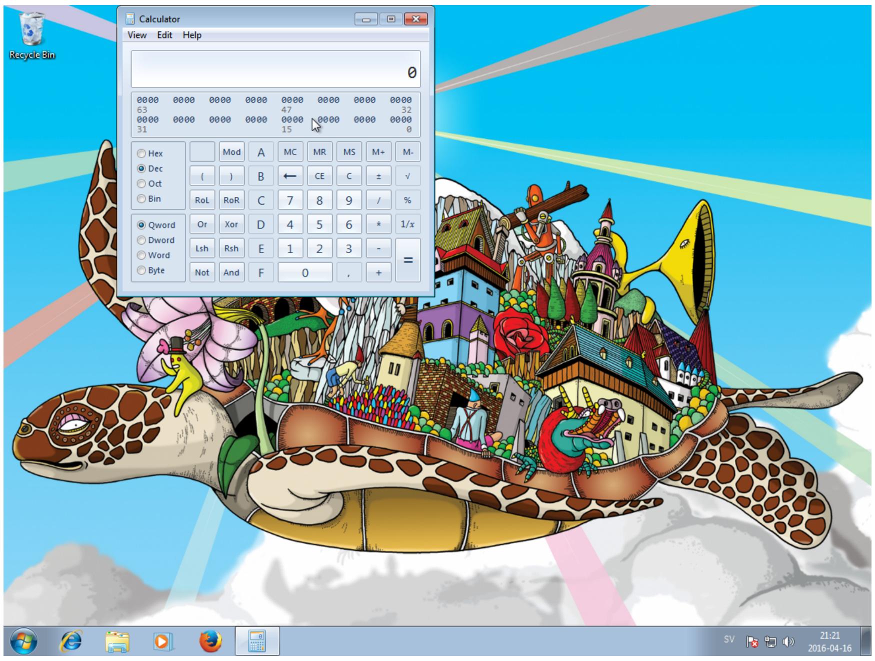


Figure 2.5: Reference screenshot

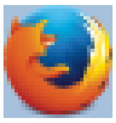


Figure 2.6: Reference template. For the sake of presentation the template in figure 2.6 has been scaled up. During testing it was the same size as the Firefox icon in the screenshot.

Cross Correlation

As with all the similarity-measures, OpenCV has implementations of both the regular and normalised variant [2, 8]. For an in-depth explanation of how normalisation *cross correlation* can be efficiently perform, J. P. Lewis have an excellent paper where the technique is explained [9].

The regular cross correlation algorithm used in OpenCV is defined as the following function:

$$\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y')) \tag{2.15}$$

where $T(x',y')$ is the specific pixel of the template and $I(x+x',y+y')$ is the corresponding pixel on the image. The x and y of I are the location on the image where the window is located.

Normalised version:

$$\frac{\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{2.16}$$

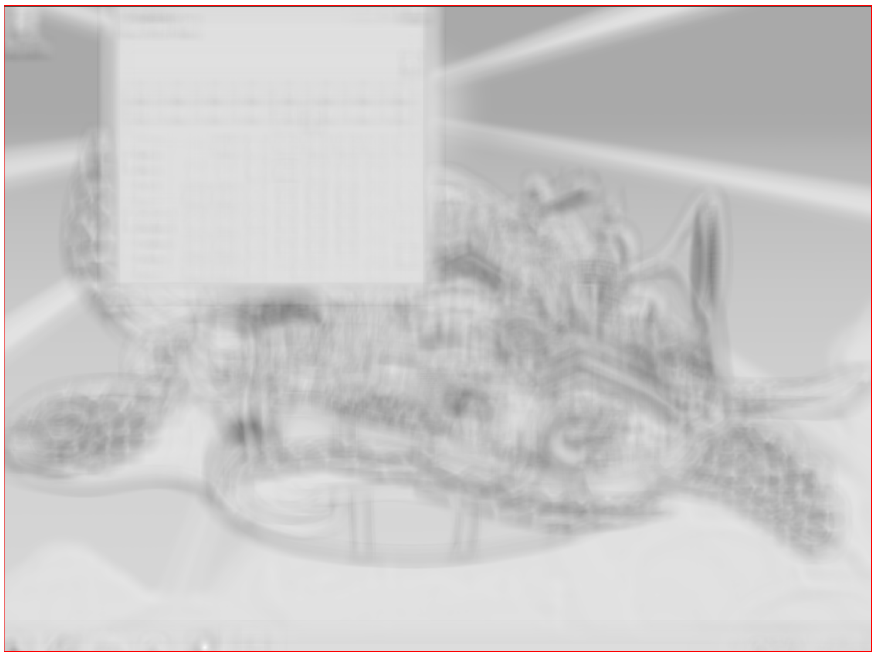


Figure 2.7: Result from *Normalised cross correlation* applied to 2.5 with template 2.6. Brighter means better match.

Squared Difference

The most significant difference between cross correlation and squared difference is that the squared difference algorithm results in the best match where the similarity score is close to zero instead of as high as possible [2, 8].

$$\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2 \tag{2.17}$$

Normalised version:

$$\frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{2.18}$$

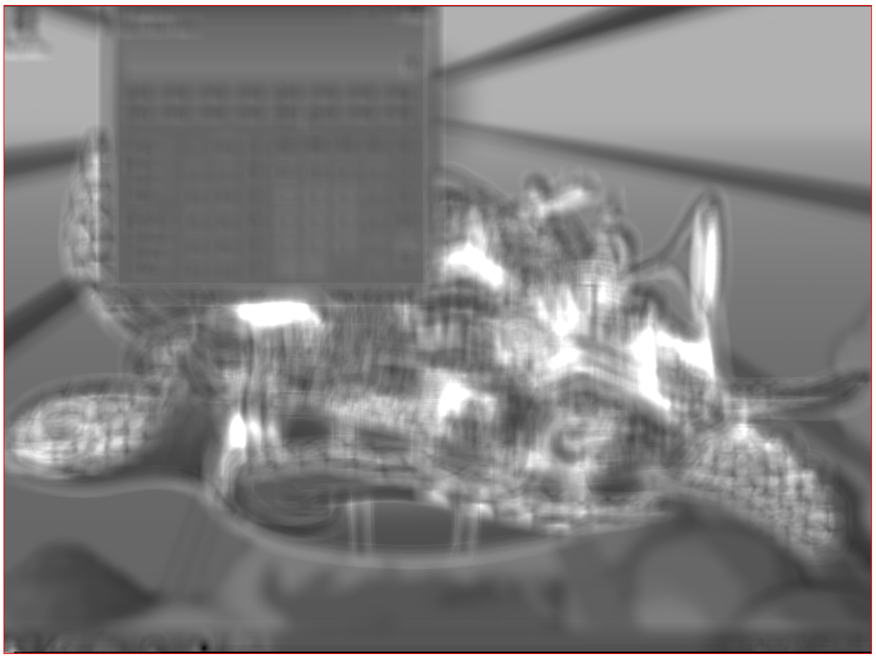


Figure 2.8: Result from *normalised squared difference* applied to 2.5 with template 2.6. Darker means better match.

Cross coefficient correlation

Cross coefficient correlation subtract the average pixel value from each pixel that is compared, resulting in a dampening of each pixels contribution and a smaller standard deviation in the result matrix [2, 8].

$$\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y')) \tag{2.19}$$

where $T'(x',y')$ and $I'(x+x',y+y')$ is equal to

$$T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'') \tag{2.20}$$

$$I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'') \tag{2.21}$$

w and h are the height and width of the template image. The I' and T' functions are the average pixel value of the template and subsection of the image respectively.

Normalised version:

$$\frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y\,\prime)^2}} \tag{2.22}$$

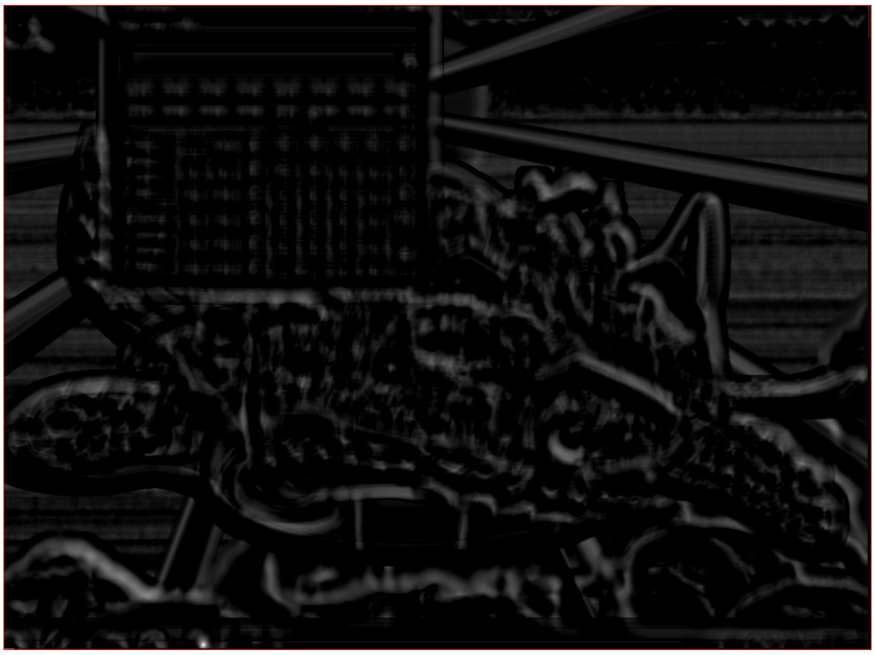


Figure 2.9: Result from *normalised cross coefficient correlation* applied to 2.5 with template 2.6 . Brighter means better match.

Canny Edge Detect

Canny Edge Detect is an algorithm described by Canny [10] as a method for finding contours in images. Canny uses a convolution matrix which is a small convoluted over the image. The convolution operation with a kernel size of 3 can be seen in equation 2.23.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = (i*1) + (h*2) + (g*3) + (f*4) + (e*5) + (d*6) + (c*7) + (b*8) + (a*9) \tag{2.23}$$

The first pass involves a convolution kernel weighted from a Gaussian blur function:

$$K = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 5 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * A \tag{2.24}$$

where A is 5x5 window of the target image.

After the Gaussian blur has been performed two more kernels are used. The kernels are used as weights. The operation is called a Prewitt operator [11].

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * K \tag{2.25}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * K \tag{2.26}$$

The result is a gradient from which the angle of an edge can be extracted. The angle is rounded to 0, 45, 90, or 135 degrees.

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.27}$$

$$\theta = atan2(\frac{G_y}{G_x}) \tag{2.28}$$

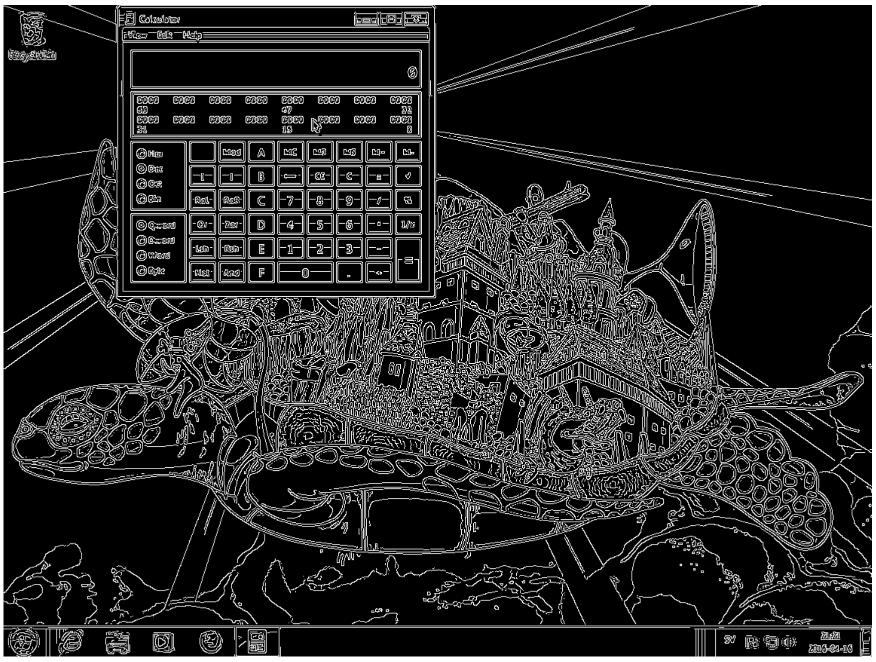


Figure 2.10: The result of applying *Canny edge detect* on 2.5.

Sikuli

Sikuli is a GUI testing automation software tool [12]. It uses OpenCV to analyse screenshots and interacts with the GUI environment using the Java's Robot class. On the *Windows* operating system, there exists a native scripting language to control GUI:s called *AutoIT* [13]. Because of the platform dependency of this scripting language, *Sikuli* can be seen considered a platform independent GUI scripting system.

Sikuli uses an extension of the OpenCV template matching algorithm in order to identify GUI objects [14]. To allow for detection of the same object but with different scales a technique called image pyramids is implemented. By using image pyramids [1] the framework gains a certain level of template size invariance.

2.2 Intelligent Agents

Within Artificial Intelligence an *Intelligent Agent* is a system that in some regards can operate autonomously, usually based on a given task and with regards to the environment around the agent. Russel and Norvig (2014) [15] groups intelligent agents into five categories:

- Simple reflex agents that only take the current sensory input as basis for its decisions.

[1] `http://docs.opencv.org/2.4/doc/tutorials/imgproc/pyramids/pyramids.html`

- Model-based reflex agents that contains an internal model of the environment. This results in an agent that can take previous actions into account when reasoning.
- Goal-based agents does not have static decision-making systems but instead evaluates possible actions in order to find one that moves it closer to a goal state.
- Utility-based agents improve further from the goal-based agents by not only thinking in world states as goal or not goal but are also able to quantify how much of a goal a certain state is and the probability that that state moves them further towards a goal state.
- Learning agents explores the state space and construct an internal model based on the findings. Exploration can be performed autonomous or through the help of a supervisor.

These categories represent an increasing scale of complexity in the agent model architecture. In this project the *model-based* agent will serve as framework. It has to be able to make basic decisions based on given information and has to be somewhat adaptable. It only has to be able to perform pre-defined tasks and perform simple *state space traversal*. The agent complexity increases when it has to interact with the GUI of the virtual environment. It should be able to find the correct GUI elements such as buttons and text fields. Even in scenarios where target elements are partially obstructed. Taking theses specification into account the agent will land somewhere between a model-based agent and a goal-based.

2.3 Statistical analysis

Binary classifier

Binary classifier is a classifier that given an input signal, outputs either a 'true' or a 'false' [16]. When evaluating a binary classifier an analysis tool called *confusion matrix* may be used. A *confusion matrix* represents all the possible classifications on one axis and the true outcomes on the second. Table 2.1 shows the *confusion matrix* of a binary classifier.

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (Tp)	False Negative (Fn)
Actual Negative	False Positive (Fp)	True Negative (Tn)

Table 2.1: Confusion Matrix for a binary classifier where the X-values are what the classifier gave as output and the Y-values are the actual value of the input.

Common metrics for evaluating a binary classifier is to calculate different rations from the confusion matrix give the problem studied [16]. If it is important for the classifier to correctly predict true results the *true positive rate*, as defined in 2.29 can be used.

$$\frac{Tp}{Tp + Fn} \tag{2.29}$$

The *true positive rate* indicates the ratio of correctly positively identified cases in relation to the incorrectly negatively identified cases. In medical testing this would constitute the ratio of people with a disease in relation to all who has the disease [17]. The *true positive rate* is often called *sensitivity* or *recall* [17, 16].

The *true negative rate* is defined as the equation 2.30.

$$\frac{Tn}{Tn + Fp} \tag{2.30}$$

which is defined as the ratio of correctly negatively identified cases in relation to all negative cases. In medical testing this ratio would constitute all people who don't have a disease in relation to all who are incorrectly diagnosed as carrying a disease [17].

In some scenarios the *precision* of a classifier is of interest. *Precision* of a classifier is defined as the equation 2.31:

$$\frac{Tp}{Tp + Fp} \tag{2.31}$$

indicating the ratio of correctly identified positive cases in relation to the incorrectly identified negative cases. In medical testing this ratio would be the same as the ratio of people correctly identified as sick in relation to the people incorrectly identified as sick.

To examine the overall performance of a binary classifier *accuracy* can be used. *Accuracy* is defined as the equation 2.32.

$$\frac{Tp + Tn}{Tp + Fp + Tn + Fn} \tag{2.32}$$

in terms of medicine *accuracy* constitutes the ratio of correctly diagnosed people in relation to the incorrectly diagnosed ones.

Receiver operating characteristic

Receiver operating characteristic plot (henceforth referred to as a ROC-curve) is a graphical way of representing the performance of a binary classifier with regards to a specific threshold value [16]. A ROC-curve is plotted with the *sensitivity* on the Y-axis and the *false positive rate* calculated by $1 - specificity$ on the X-axis. ROC-curves provide the relationship of the *sensitivity* and the *false positive rate* of a *binary classifier* given a specific values of the binary classifier discrimination threshold. An example of a ROC-curve can be observed in figure 2.11. The green line in 2.11 indicates the result of a uniformly random binary classifier.

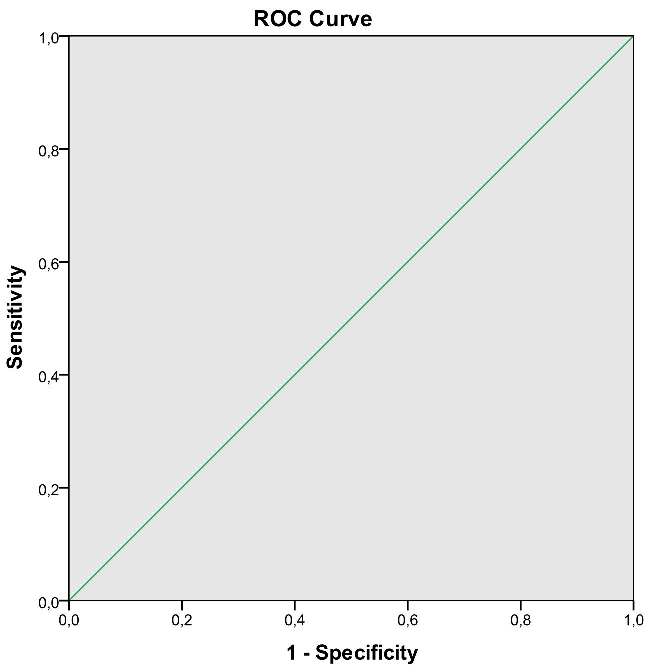


Figure 2.11: Example of a ROC-curve with a green reference line.

Area under ROC-curve
ROC-curves can be used for performance comparison of multiple binary classifiers by using the area under each respective *binary classifiers* curve [18, 19]. The area under a ROC-curve (referred to as *AUC* can be retrieved by calculating the integral under the curve. The AUC may be in the range $area \in \mathbb{R}\{0 - 1\}$ where 0 means only incorrectly classified cases and 1 means only correctly classified cases. 0.5 is the value that would be expected of a uniform stochastic binary classifier.

Logistic Regression

Most of the data analysed in this project are categorical and binary while the predictor variables are both ordinal, nominal, and continuous. As a result of the outcome data being binary, linear regression is not a suitable analysis method. Instead logistic regression will be used [20]. In simple linear regression we try to find a linear estimator optimally fitting a specific data set. This means we try to find a linear function with the least summarised distance between the data set and the fitted line.

$$Y = \beta_0 + \beta_1 x + \epsilon \tag{2.33}$$

When we try to find a best fitted model for binary data we instead try to find a function whereas the predicted mean outcome of the dependent variable is a linear function of the independent variable. For this purpose the logit function, described in equation 2.3 can be used:

$$p(x) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}} \tag{2.34}$$

By relating the value of x to the probability p(x) through the logit function the odds of p(x) = true for the different values of x can be calculated using:

$$\frac{p(x)}{1 - p(x)} = e^{\beta_0 + \beta_1 x} \tag{2.35}$$

In most cases β_0 and β_1 are estimated using the maximum likelihood function [20].

3 Method

This chapter is segmented into two parts. The first describes the method used to test template matching algorithms and their reliability in synthetic GUI environments. The second part presents how template matching will be used to determine state of the environment. Feature matching strives to find features robust to different kinds of image noise that might occur when you are tracking objects in the real world. In GUI environment many objects have a very similar look. The signature difference between two GUI objects are often lesser than the noise threshold for many feature matching algorithms. Most feature detectors are designed for 3D environments where the objects changes perspective and you very rarely have 100% percent of an object represented without noise in a single picture. Because of these reasons this thesis will not attempt to use feature detection for feature matching and instead focus on template matching. In figure 3.1 an example of the similarity between typical GUI objects.

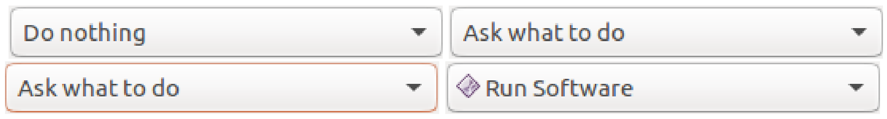


Figure 3.1: Example of the difference between buttons in a GUI environment.

3.1 Evaluation of template matching algorithms

In OpenCV the function for template matching returns a result matrix where each element represent the correlation value for that position of the template. For cross correlation and coefficient correlation you want the element of the result matrix with the highest value. For squared difference you want the lowest. The position for the best potential match is compared to the actual position of the true match for each test case. If the best potential result falls within the bounds of the true answer the test case is considered a pass. Only the normalised versions of the template matching algorithms are tested because the state identification test 3.5 requires results between 1 and 0 in order to determine positive/negative matches.

Each test case consists of two images. The first image is the templates that should be found in the second image, containing a screenshot. The correct area of the screenshot portraying the object will be compared to the most likely location provided by template matching. If the mid point of the area returned by template matching is located within the correct area the test is considered a pass.

Three different pre-processing image filters are used for this study. The first is plain *greyscale* (referred to as GRAYSCALE) and the default filter used for template matching, the second is based on the *canny edge-detect* algorithm (referred to as CANNY) described in the theory and the final is a colour washing filter designed and implemented specifically for this study which removes all colour from an screenshot that isn't in the template (referred to as FLUSH). An example of *FLUSH* can be observed in the figure 3.2

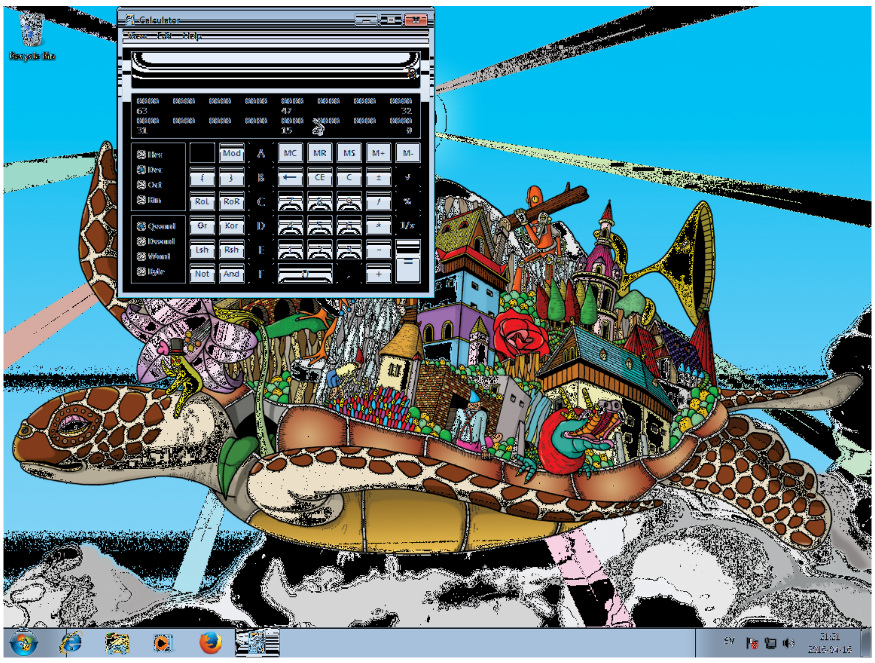


Figure 3.2: Result image after the FLUSH image filter have been applied. The template used are 2.6

The implementation of *FLUSH* can be viewed as the following code snippet. The code is simplified to better showcase the algorithm. To view the full implementation used in this study, please see the source code.

Listing 3.1: Simplified code of how the FLUSH image filter is implemented.

```
void flush_image_filter(matrix screenshot, matrix templ)
{
  screenshot_grayscale = convert_to_grayscale(screenshot);
  template_grayscale = convert_to_grayscale(templ);
  map<uchar,bool> lookup_table;
  for(uchar element: template_grayscale)
  {
    lookup_table[element] = true;
  }
  for(int x = 0; x < screenshot.columns; ++x)
    for(int y = 0; y < screenshot.rows; ++y)
    {
      if(lookup_table.contains_key(screenshot_grayscale.get_element(x,y)))
      {
        screenshot.set_element(x,y,{0,0,0});
      }
    }
}
```

3.2 Template Localization - General Reliability

This study aims to find out what environmental nose impacts template matching's ability to correctly localise a template.

The focus is on the reliability of template matching in general GUI environment and are therefore OS independent. For the same reason the partially obstructed templates are generalised from actual possible overlaps in the GUI environment (mouse pointer for example) to an amount of the feature obscured. This test also ties to find the average success rate of template matching given the different parameters.

Independent variables and values that will be examined

1. Pre-processing {GRAYSCALE, EDGE (CANNY), FLUSH }
2. Algorithm {CCORR, CCOEFF, SQDIFF }
3. Colour variance {-90, -45, 0, 45, 90 }
4. Brightness variance {0.25, 0.5, 1.0, 2.0, 4.0}
5. Obstruction {0, 25, 50, 75}
6. Template size variance {0.50, 0.75, 1.0, 1.5, 2.0}
7. List of templates
8. Template generality {extracted - same, extracted - similar, internet}

Pre-processing
The *pre-processing* parameter represents the three different colour filters which has been described in the theory and the method.
Algorithm
The *algorithm* parameter represents the three different template matching algorithms described in the theory in section: 2.1.
Colour variance

The *colour variance* parameters represents the screenshots colour wheel offset. In figure 3.3 exemplifies how the different values for this parameter would alter the test case.

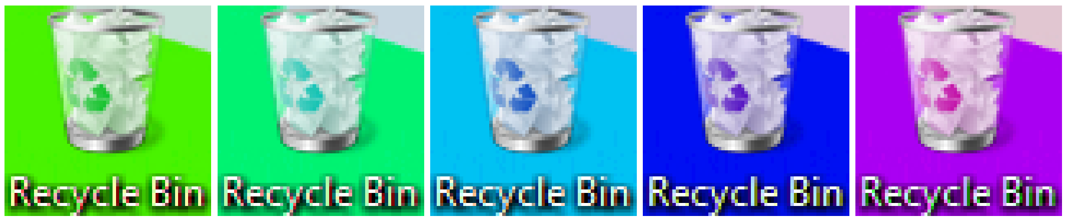


Figure 3.3: Example of how *colour variance* affects a image for the values -90, -45, 0, 45. 90 degrees respectively

Brightness variance
The *brightness variance* parameter represents a specific scalar value which all the elements in the screenshot matrix is multiplied with. This results is a change of brightness in the screenshot compared to the source image. Example of this effect can be seen in figure 3.4.

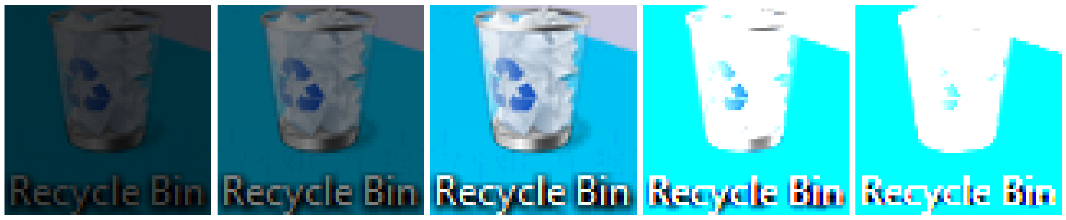


Figure 3.4: Example of how *brightness variance* affects an image for the values 0.25, 0.5, 1.0, 2.0, 4.0 respectively

Obstruction The *obstruction* is included to emulate situations where objects are partially obstructed, for example with a mouse pointer. Instead of using multiple templates representing different amount of obstruction, the obstruction will be created during run-time by placing a colour filled circle in the middle of the used template. The value of this parameter indicates the size of the white circle where the radius is calculated as $radius = width/2 * value$ where $value$ is the value of the parameter.

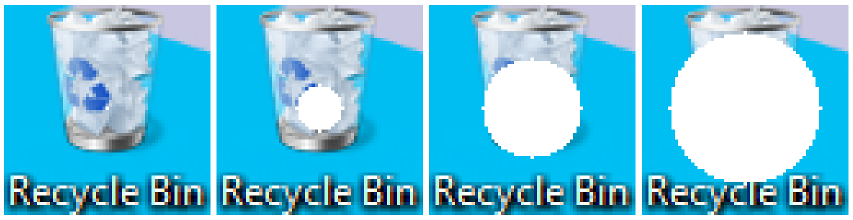


Figure 3.5: Example of how *obstruction* affects a image for the values 0, 25, 50, 75 respectively

Template size variance
In many scenarios, the object you are looking for aren't guarantee to be the same size as your template. As a result will we be testing how well template matching can handle this. As with the parameter 3.2 the size of the templates will be generated during run-time. This parameter can be observed in figure 3.6

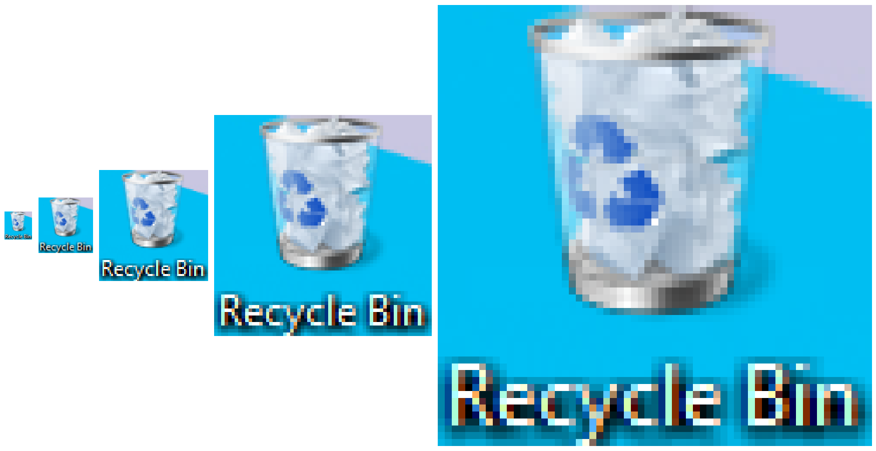


Figure 3.6: Example of how *template size variance* affects a image for the values 0.25, 0.50, 1.0, 2.0, 4.0 respectively

3.3 Template Matching - Robustness between Operating System Versions

This test aims to provide answers to how version change for an OS affects reliability of template matching. The test will only take place between the Windows version 10 and 7. This because there was a significant change in GUI aesthetics between Windows 7 and 10. This test is not targeted towards identifying image noise sensitivity (as in the previous test) for the template matching algorithms and more towards stylistic change in the templates. Therefore inducing colour, brightness, size or obstruction is not a priority. The screenshots already provide that kind of modification.

Independent variables and values that will be examined

1. Pre-processing {GRAYSCALE, EDGE (CANNY), FLUSH }
2. Algorithm {CCORR, CCOEFF, SQDIFF }
3. Version of template {Windows 7, 10 }
4. Version of environment {Windows 7, 10 }
5. List of templates

3.4 Template Matching - Computational Time

This study aims to provide answer to the question about how the agent will perform in regards to computational time. The times will be measured while running all the other tests.

3.5 State Identification

The agent architecture will follow similar design to that of a model-based agent as proposed by Russel and Norvig (2003). The architecture is designed with two primary purposes: provide structure for the agent to perform the tasks designated to it and to supply the data the agent needs to make its decisions. For this purpose a model based on classical search will be used whereas each state is represented as a set of templates that should be present in the environment. The states are linked together with the actions required to take the agent form one state to another.

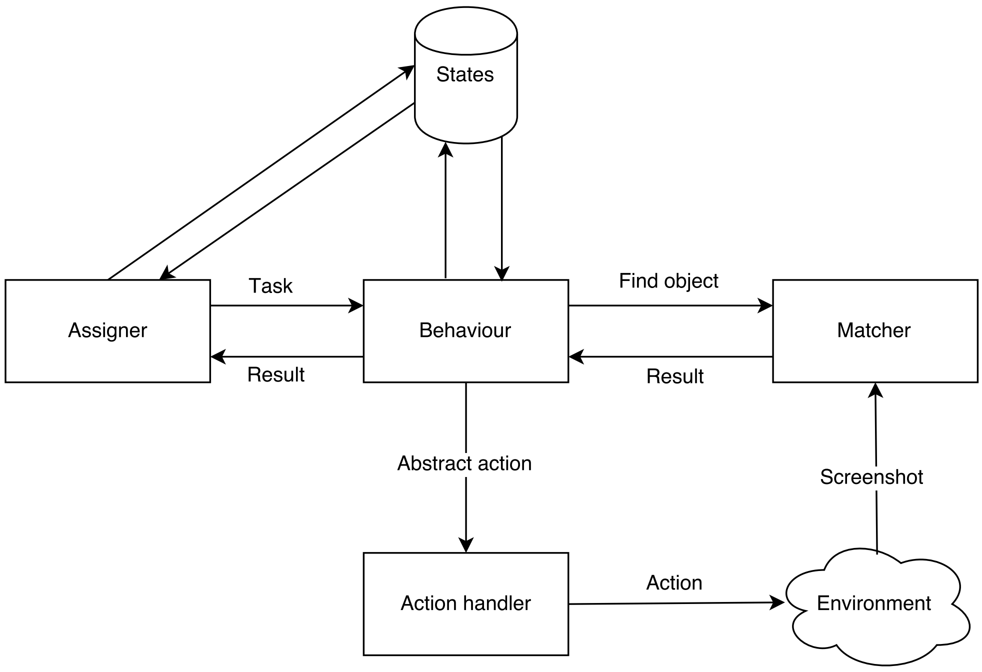


Figure 3.7: Modular overview of the proposed agent.

For the purpose of this thesis, only the *matcher* and *state* representation modules are evaluated. States are defined as a set of objects to be positively identified in the environment and the actions leading from a specific state to another.

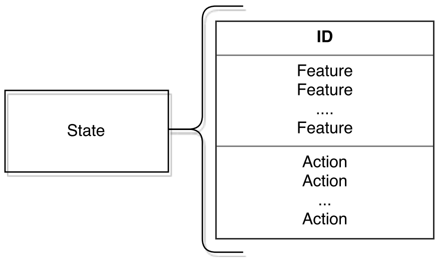


Figure 3.8: Structural definition of a state

Evaluation of the state identification reliability will be done through synthetic testing similar to how template matching were evaluated. First a set of screenshots and states will be defined. Each screenshot will represent a set of states known to be true. The agent will be asked to find the different states the different screenshots and decide if the state exists or not. The result is compared to the true answer and if the answers are the same the test case will be evaluated as a 'pass'. This tests aims to answer if the agent can (reliably) use template matching algorithms as base for its decisions.

4 Results

4.1 Template Localization - General Reliability

The result data from to study *Template Localization - General Robustness* is analysed through *logistic regression* and displayed as scatter-plots with mean predicted probability on the y-axis and the different categories of each parameter on the x-axis. The mean predicted probability indicates the mean probability of a positive (true) result of the independent variable. All tests are grouped by pre-processing image filter. The lines are interpolated for the mean predicted probability plots to better visualise the relationship between the different categories.

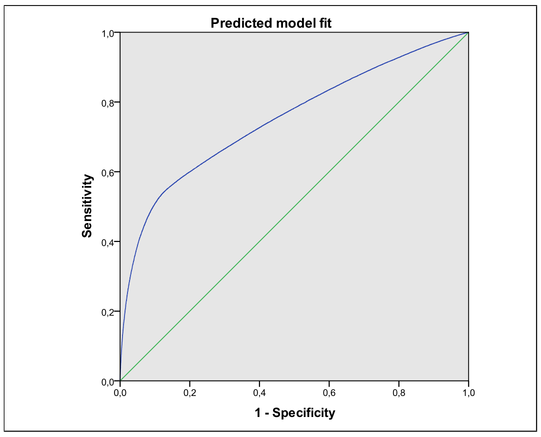


Figure 4.1: Prediction regression model for experiential data.

In figure 4.1 a *ROC-curve* over how well the regression model fits the data set. The *AUC* is equal to 0.74. A perfect model fit would result in a *AUC* of 1.0, and a random model fit would result in a *AUC* of 0.5.

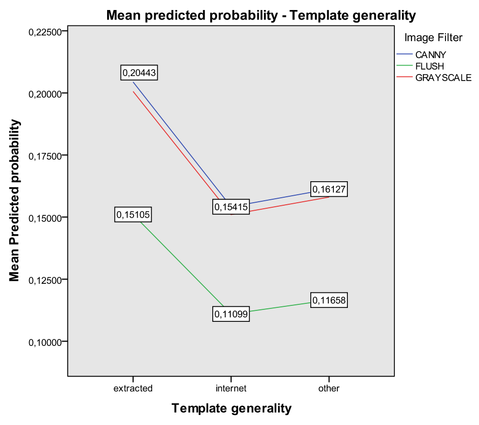


Figure 4.2: Mean predicted probability for the independent variable *template generality*. 1 is good and 0 is bad.

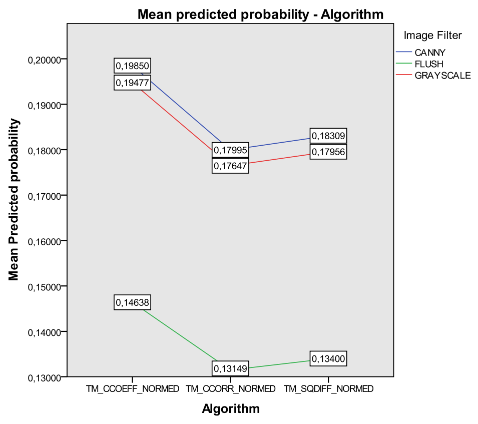


Figure 4.3: Mean predicted probability for the independent variable *algorithm*. 1 is good and 0 is bad.

Figure 4.2 shows the mean predicted probability of a positive result given the different generalities of templates. Figure 4.3 shows the mean predicted probability of a positive result given the different template matching algorithms used in the testing.

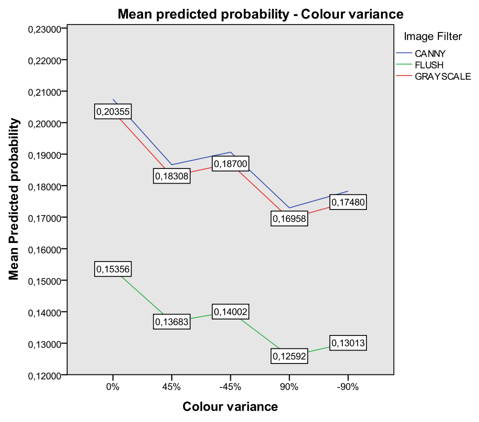


Figure 4.4: Mean predicted probability for the independent variable *colour variance*. 1 is good and 0 is bad.

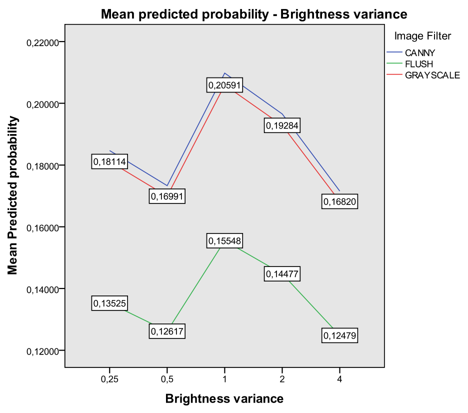


Figure 4.5: Mean predicted probability for the independent variable *brightness variance*. 1 is good and 0 is bad.

Figure 4.4 shows the mean predicted probability of a positive result given the difference in colour variance between the screenshot and template. The higher the value, the higher probability of it resulting in a correct result. Figure 4.5 shows the mean predicted probability of positive identification when the brightness of the screenshot is varied. The higher the value, the higher probability of it resulting in a correct result.

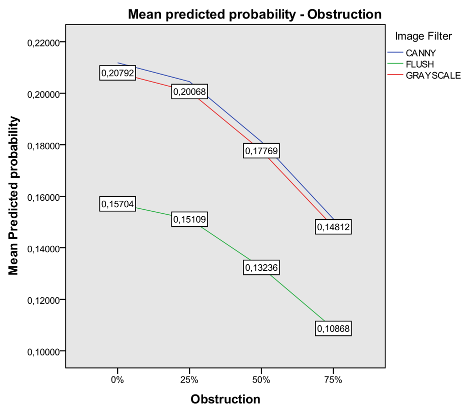


Figure 4.6: Mean predicted probability for the independent variable *obstruction*. 1 is good and 0 is bad.

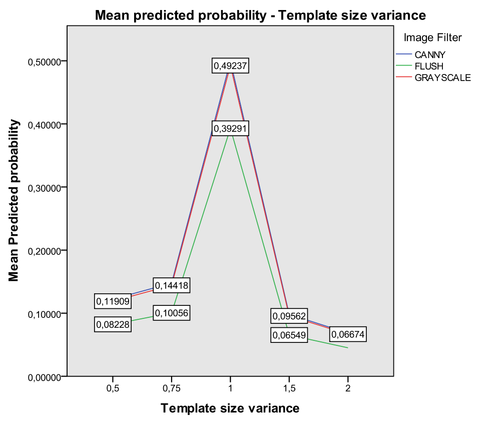


Figure 4.7: Mean predicted probability for the independent variable *template size variance*. 1 is good and 0 is bad.

Figure 4.6 the mean predicted probability of positive identification when the template is obstructed to a certain percent. The higher the value, the higher probability of it resulting in a correct result. Figure 4.7 shows the mean predicted probability of positive identification when the size of the template is offset by a given scale. The higher the value, the higher probability of it resulting in a correct result.

Case Summaries

	Template generality	Algorithm	Image Filter	Colour variance	Brightness variance	Obstruction	Template size variance	Mean success rate	# of cases
1	extracted	TM_CCOEFF_NORMED	FLUSH	0%	1.0	0%	1.0	1.00	41
2	extracted	TM_CCOEFF_NORMED	GRAYSCALE	45%	0.25	0%	1.0	1.00	41
3	extracted	TM_CCOEFF_NORMED	GRAYSCALE	90%	1.0	0%	1.0	1.00	41
4	extracted	TM_CCOEFF_NORMED	GRAYSCALE	0%	0.25	0%	1.0	1.00	41
5	extracted	TM_CCOEFF_NORMED	GRAYSCALE	0%	0.5	0%	1.0	1.00	41
6	extracted	TM_CCOEFF_NORMED	GRAYSCALE	0%	1.0	0%	1.0	1.00	41
7	extracted	TM_CCOEFF_NORMED	GRAYSCALE	45%	0.25	0%	1.0	1.00	41
8	extracted	TM_CCOEFF_NORMED	GRAYSCALE	45%	0.5	0%	1.0	1.00	41
9	extracted	TM_CCOEFF_NORMED	GRAYSCALE	45%	1.0	0%	1.0	1.00	41
10	extracted	TM_CCORR_NORMED	GRAYSCALE	0%	0.25	0%	1.0	1.00	41
11	extracted	TM_CCORR_NORMED	GRAYSCALE	0%	0.5	0%	1.0	1.00	41
12	extracted	TM_CCORR_NORMED	GRAYSCALE	0%	1.0	0%	1.0	1.00	41
13	extracted	TM_CCORR_NORMED	GRAYSCALE	45%	0.25	0%	1.0	1.00	41
14	extracted	TM_CCORR_NORMED	GRAYSCALE	45%	0.5	0%	1.0	1.00	41
15	extracted	TM_CCORR_NORMED	GRAYSCALE	45%	1.0	0%	1.0	1.00	41
16	extracted	TM_SQDIFF_NORMED	GRAYSCALE	45%	1.0	0%	1.0	1.00	41
17	extracted	TM_SQDIFF_NORMED	GRAYSCALE	0%	1.0	0%	1.0	1.00	41
18	extracted	TM_SQDIFF_NORMED	GRAYSCALE	45%	1.0	0%	1.0	1.00	41
19	extracted	TM_CCOEFF_NORMED	FLUSH	90%	1.0	0%	1.0	0.98	41
20	extracted	TM_CCOEFF_NORMED	GRAYSCALE	45%	0.5	0%	1.0	0.98	41

Table 4.1: Top 20 parameter configurations in terms of mean success rate. *Mean success rate* is the key column where 1 is good and 0 is bad.

Table 4.1 shows the top 20 best parameter configurations in terms of *accuracy*. The column *mean success rate* shows the *accuracy* of said parameter configuration and # *of cases* shows the amount of test cases that have been tested for each configuration.

4.2 Template Localization - Robustness between Operating System Versions

In this section the result from the second study concerning the robustness of localising templates over different versions of an operating system is presented. The results are the means success rates given different independent variables used in the study.

Case Summaries

	Algorithm	Image Filter	Mean Success rate	# of cases
1	TM_CCOEFF_NORMED	GRAYSCALE	71	38
2	TM_CCORR_NORMED	GRAYSCALE	66	38
3	TM_SQDIFF_NORMED	GRAYSCALE	63	38
4	TM_CCORR_NORMED	CANNY	58	38
5	TM_CCOEFF_NORMED	CANNY	55	38
6	TM_SQDIFF_NORMED	CANNY	53	38
7	TM_CCOEFF_NORMED	FLUSH	47	38
8	TM_CCORR_NORMED	FLUSH	32	38
9	TM_SQDIFF_NORMED	FLUSH	32	38

Table 4.2: Mean success rate in terms of *accuracy* over all configurations of algorithms and image filters. *mean success rate* is the key column. 1 is good and 0 is bad.

Table 4.2 shows the performance of localising template cross OS version. The key column to examine is the *mean Success rate* column which indicates the *accuracy* for that set of parameter values. The # *of cases* indicates how many different test cases are tested using the relevant parameter configuration.

4.3 Template Matching - Computational Time

This section focuses on presenting the mean computational time required by the different pre-processing filters and template matching algorithms used in the study.

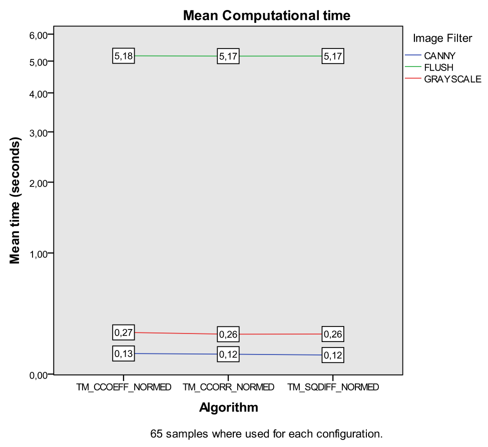
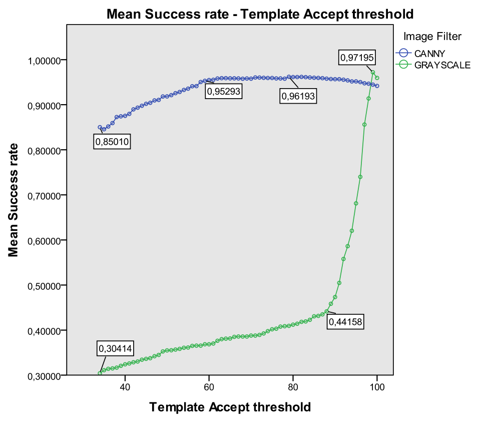


Figure 4.8: Mean computational time for *image filter* and *algorithm*. Values closer to zero is good.

The result can be viewed in the figure 4.8 where the different template matching algorithms are presented on the x-axis and the mean computational time on the y-axis. The data is grouped by pre-processing image filters and the lines are interpolated to provide a easier visualisation of the data.

4.4 State Identification

This section focuses on displaying the result of the study testing the agents ability to discern what states the world currently inhabits. The result is shown as scatter-plots over the different threshold-variables tested and the algorithms used in the study. All data is grouped by the pre-processing image filter used in the study and lines are interpolated to better visualise the result.



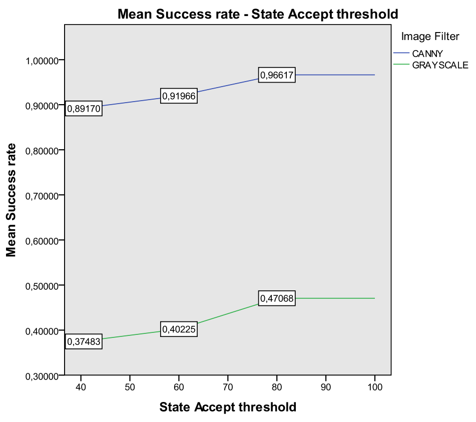


Figure 4.9: Mean success rate for the independent variable *template Accept threshold*. Y-values close to 1 are good and values close to 0 is bad.

Figure 4.10: Mean success rate for the independent variable *state accept threshold*. Y-values close to 1 are good and values close to 0 are bad.

Figure 4.9 displays the mean success rate in terms of *accuracy* of the agent correctly identify the current states existing in the environment. Higher values are better. Figure 4.10 displays the mean success rate of the agent correctly identifying states given the amount of templates in a state that needs to be positively identified. The ratio is calculated the same way as 4.9 with the formula presented in the equation mentioned above.

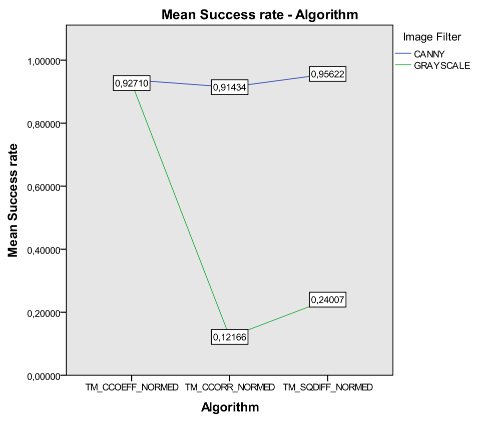


Figure 4.11: Mean success rate for the independent variable *algorithm*. Y-values close to 1 is good and values close to 0 are bad.

Figure 4.11 displays the mean success rate of the agent correctly identifying states given the amount of templates in a state that needs to be positively identified.

Case Summaries

	Algortihm	Image Filter	State Accept threshold	Template Accept threshold	Mean Success rate	# of cases
1	TM_CCOEFF_NORMED	GRAYSCALE	80	99	1.00000	324
2	TM_CCOEFF_NORMED	GRAYSCALE	100	99	1.00000	324
3	TM_CCOEFF_NORMED	GRAYSCALE	80	98	1.00000	324
4	TM_CCOEFF_NORMED	GRAYSCALE	100	98	1.00000	324
5	TM_CCOEFF_NORMED	GRAYSCALE	80	97	1.00000	324
6	TM_CCOEFF_NORMED	GRAYSCALE	100	97	1.00000	324
7	TM_CCOEFF_NORMED	GRAYSCALE	80	96	0.99690	324
8	TM_CCOEFF_NORMED	GRAYSCALE	100	96	0.99690	324
9	TM_CCORR_NORMED	GRAYSCALE	80	99	0.99380	324
10	TM_CCORR_NORMED	GRAYSCALE	100	99	0.99380	324
11	TM_SQDIFF_NORMED	GRAYSCALE	80	99	0.99380	324
12	TM_SQDIFF_NORMED	GRAYSCALE	100	99	0.99380	324
13	TM_SQDIFF_NORMED	GRAYSCALE	80	98	0.99380	324
14	TM_SQDIFF_NORMED	GRAYSCALE	100	98	0.99380	324
15	TM_CCOEFF_NORMED	GRAYSCALE	80	95	0.99380	324
16	TM_CCOEFF_NORMED	GRAYSCALE	100	95	0.99380	324
17	TM_CCOEFF_NORMED	GRAYSCALE	80	94	0.99380	324
18	TM_CCOEFF_NORMED	GRAYSCALE	100	94	0.99380	324
19	TM_CCOEFF_NORMED	GRAYSCALE	80	93	0.99380	324
20	TM_CCOEFF_NORMED	GRAYSCALE	100	93	0.99380	324

Table 4.3: Top 20 parameter configurations. *Mean success rate* is the key column and values close to 1 is good and values under 0.95 are bad.

Table 4.3 shows the best 20 parameters configurations achieved in this study. The key column to examine is the *mean success rate* column which indicates the *accuracy* for that set of parameter values. The # *of cases* indicates how many different test cases are tested using the relevant parameter configuration.

5 Discussion

In this study we explored whether artificial agents simulating user activity in a virtual network could use computer vision algorithms in order to perform actions such as interacting with the environment and determining the state of the GUI environment. The study focused on evaluating the computer vision algorithm *template matching* combined with different image filters in order to determine their suitability for the task with a focus on reliability.

Evaluation of *template matching* were subdivided into four main studies. The first study aimed at evaluating what type of image noise reduced the algorithms ability to localize a given object in the environment. This study also served a secondary purpose, to provide insight into what is required from both the algorithms and the environment to provide a reliable result.

The second study presented real scenarios where objects were to be localized over the bounds of operating system versions. The purpose of this study was to provide support for the first study by not simulating image noise and instead provide real scenarios.

Although no maximum-computation time were given as requirement for the study, it was of interest to analyse this aspect of *template matching* and the image filters. This was analysed in the third study.

The fourth and final study targeted the agent's ability to determine what state the environment inhabited at a given moment. States were defined as set of objects that had to exist in the environment for the state to be considered true and that the environment has to be in a temporally stable state.

5.1 Object Localization

The result of the first study were evaluated by using logistic regression the regression model fitted the data with an accuracy of 74%, which can be seen in figure 4.1.

Noise Robustness

Robustness to image noise was one of the main aspects of the problem that we wanted to analyse in this thesis. In the following paragraphs the result showcased in the figures under section "Template Localization - General Reliability" will be discussed and analysed.

In this study four primary image noises were tested: colour variance, brightness variance, partial obstruction, and template scale variance.

The logistic regression result for *colour variance* can be viewed in the figure 4.4. The result indicates that no colour variance results in the best object localization results and progressively worsens the more variance that is introduced. Varying the colour clockwise on the colour wheel results in a better result but the difference is less than 1% and can be considered not statistically significant.

The result of testing *brightness variance* can be seen in the figures 4.5. Similar result to *colour variance* can be observed where the further from the "zero variance" value. The plot indicates that doubling the brightness is generally better than halving it and should result in only about 0.8% decreased performance compared to no variance. Reducing the brightness to a fourth perform on average 1% better than not reducing the brightness to a half.

Obstruction was tested by generating filled white circles on the templates which in turn was used for the object localisation. The result this parameter affected the result can be seen in figures 4.6. Inspection of said figure indicates that no obstruction results in the best mean value with a mean predicted probability of 0.2. Adding 25% of obstruction to the template reduces the accuracy with less than 1% but after this the performance decreases down to 0.14 in predicted probability at 75% obstruction.

Figure 4.7 showcases the effect *template scale variance* had on the study. The result points to *template scale variance* being the most significant image noise factor where even a slight deviation of size between template and screenshot results in a significant lowered probability of success. There is a significant peak at the value for "no variance" and at the first sample value in each direction the result quickly drops down to about 10% predicted probability.

Algorithms and Image Filters

Figure 4.3 displays the three different template matching algorithms grouped by the three image filters. *TM_CCOEFF_NORMED* performed the best with *TM_CCORR_NORMED* and *TM_SQDIFF_NORMED* both about 2% worse.

All the results from the logistic regression analysis has been grouped by image filter and the same trend can be observed with all the figures. *FLUSH* performs on average 5% worse than the other two. *GRAYSCALE* performed on average less than 1% worse than *CANNY*.

Template Generality

Templates extracted from the same screenshot it was tested on performed better than both templates extracted from similar screenshots and templates downloaded from the internet. This can be observed in figure 4.2. Template extracted from the same screenshot performed about 4-5% better than the other alternatives.

Best cases

The table 4.1 where similar result to what the regression model predicted can be views. The best possible parameter configuration results in a 100% success rate. This requires the *template scale offset* to equal to 0, *colour shift* either 45% or 0%, *luminance shift offset* to be equal to 0, and no *template obstruction*. In the logistic regression result *CANNY* performed slightly better than *GRAYSCALE* viewed over the entire data set but the table indicates the best results are caused by *GRAYSCALE* this indicates that *CANNY* performs on average better when there's noise in the image but worse if the noise is kept at a minimum.

Cross OS versions

In table 4.2 the result from testing object localization between two versions of the same operating system can be seen. The operating system and version chosen were Windows with version 7 and 10 where the graphical difference dramatically changed. The motivation of this is to provide a realistic "worst" case scenario compared to most operating systems. Windows 10 also uses a minimalistic design common in many modern applications in contrast to the previous gradient-heavy design [21, 22]. No configuration of image filter and template matching algorithm results in close to 100% success rate. The best configuration is *TM_CCOEFF_NORMED* with *GRAYSCALE*. This is the same configuration which was observed in the larger object localization study.

Method

The result achieved in this study fits well into other studies done in the area. The GUI automation tool-kit *Sikuli* presented in the theory uses *TM_CCOEFF_NORMED* as primary matching method [23]. *Sikuli* compensates for the template size variance by producing image pyramids. Indicating that the authors also encountered significant problems with size variance of objects.

The object localisation study was performed on already extracted screenshots and template. This fact could skew the result either by not representing a good enough cross section of the possible scenarios that might occur during run-time. Care was taken into using screenshots and template from as a diverse set of scenarios as possible. Windows 7 was chosen as the base OS because it represent a GUI style that is still very popular today even tough the minimalistic style of Windows 10 and material design has gained momentum the last couple of years.

5.2 Computational Time

Figure 4.8 shows the mean computational time for each template matching algorithm grouped by image filter. There were small difference in computational time between the template matching algorithms, at most 0.01 seconds. Between the image filters however the difference was significant with edge-detect performing in at about half the time of greyscale-matching, colour flushing required an order of magnitude more computational time. The easiest explanation of this is that the image flush filter were implemented for this thesis and therefore not optimized in the same way as the other two.

Method

The measurements for the study were taken at the same time as the *state identification* study was performed. The timer started right before the call to the matcher module was performed and stopped right after the return of said function. The function itself has a high likelihood of not being implemented optimally but the result shouldn't be many 100th of a seconds off. In total there were 65 samples for each configuration of algorithm and image filter providing an adequate ground for calculating mean computational time.

5.3 State Identification

The final study provided three graphs and one table. The graphs presented the different threshold values used by the agent to determine if a state existed in a given environment or not. The result in figure 4.9 indicates that a small difference (less than 3%) between the objects representation in the environment and the template used can result in a 100% success rate. According to figure 4.10 at least 60% of the templates defining a state has to be positively identified in order to provide a close-to 100% success rate. The different template matching algorithms displayed in figure 4.11 also affected the result with cross coefficient correlation outperforming the other two. To achieve

Method

Due to time constraints the colour washing image filter couldn't be tested on the final study. If it was included this project would go on for several weeks more than the time dedicated. Taking into account the first and second study, it's likely that it would perform similar to greyscale. The fact that such a limited selection of operating systems and colour themes tested could skew the result from what to expect in a generic scenario. Also the fact that many templates were extracted from the screenshots tested could result in a higher than average result.

The screenshots and template were once again extracted from pre-fetched screenshots and in this case many states were defined as multiple steps in the same software process. The idea behind this was that it would present a worst-case scenario where many states have very similar templates. As default *Sikuli* considers a template to be found if the similarity measure (template accept threshold) is above 0.7. This contrasts to this study where nothing can be guaranteed lover than 0.96. A possible explanation is that *Sikuli* isn't designed to look for a full database filled with template and more towards looking for specific objects defined by the programmer/tester.

5.4 Method

The best approach to implementing and testing of the agent prototype would be to create a virtual network and test the agent in a real scenario. With the artificial simulation used in this thesis practical problems such as latency, time, and unexpected environmental behaviour is hard to emulate. If more time were allocated to the project more algorithms could be studied. The values selected for each testing parameter are picked to present trends in the matching algorithm on a broad scale. Because the screenshots and templates are arbitrarily picked and from a very limited number of GUI environments for this project the results might not be representative for all OSes and GUI environment. The choice of Windows was based on the fact that the VM environment that this work was evaluated for primarily uses Windows guests.

5.5 Reproducibility & Repeatability

The algorithms used were based on OpenCV version 2.4.9 and uses the C++ variant of the library.

All raw data, including template, screenshots and test cases are available to the public. Using the same agent code, and test cases the result would be exactly the same as in this thesis. The source is published in order to allow for others to continue work on the project.

5.6 Ethical aspects

The virtual agent prototype developed in this project will be used to manage virtual computer networks. The technology developed in this project can be adapted for other virtual network systems where integrity of the virtual environment or a general management solution is desired. For example Docker [1] is a container utility to sandbox computer environment where there might be a desire to manage the containers without contaminating the virtual environment with scripts and management software.

The agent prototype is developed for a virtual environment where cyberwarfare training is taking place. Although the agent technology is generic it isn't hard to imagine militarization and deployment of the agent technology outside training environments.

[1] https://www.docker.com/

5.7 Future Work

As mentioned in the discussion, the clearest way to go forward with this study is to fully implement the agent system presented in the method and do field testing. The focus of this study was targeted at evaluating the reliability of template matching to localise and identify objects in a GUI environment. Digital image processing is a large field with many algorithms and techniques such as feature matching and colour histogram based tracking [24, 2] that might provide a more reliable solution than template matching to the problem addressed in this thesis. Because of the modularity of the agent frameworks support of other technologies can be used for the computer vision information gathering part. Template matching is not confined to only OpenCV and other libraries could also be of interest to study.

In this study a GUI state was defined as a set of templates. Testing this approach only involved examine the amount of templates positively identified in each state. This is a very simple way of reasoning about the state of the GUI environment and it would be of interest to examine more complex reasoning such as machine learning or inference [15].

6 Conclusion

In this thesis reliability of using computer vision as sensory system for a intelligent agent designed to manage a virtual computer environment has been evaluated. We present possible computer vision techniques that might provide a solution to the problem and decided on *template matching* as an interesting algorithm to examine. The examination was split up into four steps. The first involved *template matchings* reliability of localizing known objects (GUI elements) in a GUI environment, the second examination step involved measuring the time requirement for the different image filters and algorithms used, and the third step involved examining reliability of the algorithms in identifying whether objects are present in an environment.

Given the result and discussion, the final answer to the three research questions are as follow:

- *Can template matching be used to reliably identify element in a graphical user interface?*

  Given a high degree of similarity between the object in the environment and the template used, reliability closing on 100% accuracy can be achieved. This indicates that template matching can be used to reliably localize and identify elements in a typical GUI environment.

- *Can a virtual user agent use this data in order to reliably navigate the environment?*

  By defining an agent state as a set of objects that have to be positively identified in the environment, and a high degree of similarity between the objects and templates, a virtual user agent can use template matching to reliably navigate the environment.

- *What is the performance of the developed agent in terms of computational time?*

  On average the computational time of the *template matching* algorithms land on the same hundredth of a second for a single image pass, with *TM_SQDIFF_NORMED* taking on average one hundredth of a second longer. When factoring in image filter *CANNY* the total computational time lands around 1 tenth of a second, *GRAYSCALE* results in 2 tenth of a second, and *FLUSH* around 5 seconds.

Bibliography

[1] *CRATE - Cyber Range and Training Environment*. http://www.foi.se/sv/Var-kunskap/Informationssakerhet-och-Kommunikation/Informationssakerhet/Labb-och-resurser/CRATE/. Accessed 8 Feb. 2016.

[2] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Me-dia, Inc, 2008.

[3] David G. Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[4] Chris Harris and Mike Stephens. "A combined corner and edge detector." In: *Alvey vision conference*. Vol. 15. Citeseer. 1988, p. 50.

[5] Hans P. Moravec. *Obstacle avoidance and navigation in the real world by a seeing robot rover.* Tech. rep. DTIC Document, 1980.

[6] *Brute Force Matcher*. http://docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html. Accessed 23 June. 2016.

[7] *FLANN - Fast Library for Approximate Nearest Neighbour*. http://www.cs.ubc.ca/research/flann/. Accessed 23 June. 2016.

[8] *Template Matching — OpenCV 2.4.12.0 documentation*. http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html. Accessed February 8, 2016.

[9] John P. Lewis. "Fast template matching". In: *Vision interface*. Vol. 95. 120123. 1995, pp. 15–19.

[10] John Canny. "A computational approach to edge detection". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 6 (1986), pp. 679–698.

[11] Judith MS. Prewitt. "Object enhancement and extraction". In: *Picture processing and Psychopictorics* 10.1 (1970), pp. 15–19.

[12] Tom Yeh, Chang Tsung-Hsiang, and Robert C. Miller. "Sikuli: using GUI screenshots for search and automation". In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM. 2009, pp. 183–192.

[13] Jason Brand and Jeff Balvanz. "Automation is a breeze with autoit". In: *Proceedings of the 33rd annual ACM SIGUCCS conference on User services*. ACM. 2005, pp. 12–15.

[14] *GitHub.com/sikuli/sikuli*. `https://github.com/sikuli/sikuli/tree/master`. Accessed February 10, 2016.

[15] Stuart Russell and Peter Norvig. "Artificial Intelligence A modern approach". In: (2014).

[16] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern recognition letters* 27.8 (2006), pp. 861–874.

[17] Douglas G. Altman and J. Martin Bland. "Diagnostic tests. 1: Sensitivity and specificity." In: *BMJ: British Medical Journal* 308.6943 (1994), p. 1552.

[18] James A. Hanley and Barbara J. McNeil. "The meaning and use of the area under a receiver operating characteristic (ROC) curve." In: *Radiology* 143.1 (1982), pp. 29–36.

[19] Andrew P. Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.

[20] Jay Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2015.

[21] *What is Windows Aero?* `http://windows.microsoft.com/sv-se/windows-vista/what-is-windows-aero`. Accessed June 4, 2016.

[22] *Windows 10 Design: Getting the balance right*. `https://blogs.windows.com/windowsexperience/2015/04/29/windows-10-design-getting-the-balance-right/`. Accessed June 4, 2016.

[23] Chang Tsung-Hsiang, Tom Yeh, and Robert C. Miller. "GUI testing using computer vision". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2010, pp. 1535–1544.

[24] Wenmiao Lu and Yap-Peng Tan. "A color histogram based people tracking system". In: *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*. Vol. 2. IEEE. 2001, pp. 137–140.