University of Dublin

TRINITY COLLEGE

Computer Vision in User Interfaces

Darragh Hickey

B.A.(Mod.) Computer Science

Final Year Project April 2015
Supervisor: Dr. Kenneth Dawson-Howe

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

_____ _____

Name Date

Acknowledgements

I would like to thank my supervisor Dr. Kenneth Dawson-Howe for his suggestions, feedback and constant guidance throughout this project.

I would like to thank Netsoc and Douglas Temple for providing a virtual machine with which I could test and share this project.

I would also like to thank my friends, family and classmates for their support during this project and throughout the entirety of my degree.

Abstract

This report describes the implementation of a user interface controlled through a webcam with the aid of computer vision. The interface is made up of a map that the user controls through simple movements of their head and hands.

Computer vision is rarely used in the context of interfaces. The aim of this project is to investigate the effectiveness of computer vision in user interfaces and how it can be included in human computer interaction.

Contents

Acknowledgements iii
Abstract iv
Contents v
Table of Figures vii
1 Introduction 1
1.1 Aims 1
1.2 Research Objectives 1
2 Background 3
3 Technologies Used 4
3.1 OpenCV 4
3.2 Node.js 4
3.3 WebSocket 5
3.4 WebRTC 5
3.5 Google Maps JavaScript API v3 6
3.6 JQuery 6
3.7 HTML 5 Canvas 7
4 Computer Vision Techniques 8
4.1 Haar Classifiers 8
4.2 Colour Segmentation/Thresholding 10
4.3 Back Projection 11
4.4 Morphological Operations 12
4.5 Connected Components 13
5 Implementation 14
5.1 Architecture 14
5.2 Frontend 14
5.2.1 Map.js 15

5.2.2 Webrtc.js 15
5.2.3 Websocket.js 15
5.3 Node Server 16
5.4 OpenCV Program 17
5.4.1 Head and Eye Detection 17
5.4.2 Hand Detection through Colour Segmentation 19
5.4.3 Hand Detection through Back Projection 19
6 Evaluation 21
6.1 Computer Vision Processes 21
6.1.1 Face and eye testing 21
6.1.2 Hand Detection Testing – Colour Segmentation 23
6.1.3 Hand Detection Testing – Histogram and Back Projection 25
6.2 Response Time 26
7 Future Work 27
7.1 More Gestures 27
7.2 Other Applications 27
7.3 Feature Detection 27
7.4 Eye Movement Tracking 28
7.5 Move from server side to client side. 28
8 Conclusion 29
9 References 31

Table of Figures

Figure 4.1: Haar Wavelet 8
Figure 4.2: Haar-like features used by OpenCV 8
Figure 4.3: Haar-like feature applied to an image 9
Figure 4.4: HSV Colour Model 10
Figure 4.5: 2D view of YCbCr colour model, the missing dimension is the Y (lightness) dimension 11
Figure 4.6: Before (left) and after (right) for an erode followed by a dilate 12
Figure 5.1: Project Architecture 14

1 Introduction

This chapter will outline the aims, motivations and core objectives of the project.

This project presents a user interface controlled through a webcam by the movement of the user instead of through the mouse and keyboard. The interface is a map and it is controlled by the user moving their head or their hand.

The head movements can range from moving the head completely in a direction to simply rotating the head to face the point of the map the user wishes to move towards. The map's zoom level can be increased or decreased by moving the head towards the camera or away from the camera respectively. The movement of the head is judged in relation to a starting resting position chosen by the user.

The user can move the map with their hand by holding the hand up in the direction they want the map to move.

1.1 Aims

The aim of this project is to create a user interface that is controlled through a camera using computer vision to process the input. This involves a webpage containing a map as the interface. This webpage should access the user's webcam and feed the video stream input to a computer vision program.

The program should analyse the video frames and after processing should return location data for the user's head, eyes and hand to the webpage. The webpage should then use this location data to control the map, thus eliminating the need to use a mouse.

1.2 Research Objectives

This project is motivated by questions about the future of user interfaces, and whether or not computer vision has a place in user interfaces. Touch screens have seen an increase in popularity due to smart phones, but it seems major changes in how we interact with our devices only come due to necessity.

Nothing has really challenged the mouse and keyboard as our main way of controlling computers and interfaces have been built around this type of control. It is very possible that we have designed our interfaces in a way that can only be controlled by mouse and keyboard or that people are just too used to this method of interaction.

Nonetheless it is still interesting to investigate possible alternatives to the current way we interact with these machines and computer vision provides a good opportunity to do so. Webcams are very common these days and computer vision systems can be built to work with and take advantage of the natural movements of humans in order to attempt to control an interface.

The project has an overall objective of trying to find out if computer vision can be effectively used as a means for controlling computers. And if it can be, what place can computer vision hold in human computer interaction.

2 Background

This section will discuss the origin and the evolution of the project idea.

This final year project stemmed from an idea to create a user interface controlled by a webcam and a Wii Mote from Nintendo's Wii console. The Wii Mote can be essentially used as a camera as it tries to detect infra-red LEDs. The sensor bar from the Wii can be attached to a user and the Wii Mote can be set up like a camera and used to find and track the position of the user. The user can instead attach two infra-red LEDs to a hat or glasses to avoid using the sensor bar. [1]

The idea of using the Wii Mote was eventually dropped in favour of just using a standard webcam. It was an interesting concept but the Wii Mote added extra complexity in set up and in forcing the user to wear infra-red LEDs, and it doesn't offer an extreme increase in the tracking speed of a user over a normal webcam.

So with the idea settled on a webcam controlled user interface, it had to be decided what the interface would be. The idea of a video game as the interface to be controlled came up. However video games have seen substantial inclusion of computer vision in the context of controlling interfaces. This is perhaps due to often having 3D environments where computer vision works well or maybe because users are already used to abandoning mouse and keyboard for interacting with video games. It would possibly be more interesting to look at an area in which computer vision is not often used.

The idea of using a map was also raised. This was a good fit as body movement could control much of the functions needed to view a map. Map applications are typically controlled by the mouse which we are trying to reduce dependence on. With this decision the project idea was fully formed.

3 Technologies Used

This section will provide descriptions of each of the technologies used throughout this project and the reasons why each technology was used.

3.1 OpenCV

OpenCV (Open Source Computer Vision Library) is a library that supplies algorithms for computer vision and machine learning and aims to provide a common infrastructure for computer vision applications.

It was initially released in 1999, with the second release OpenCV2 released in 2009. It is written in C++ which is also its primary interface. It also has interfaces for Java, Python and C. The C++ interface was used during this project to create a computer vision program.

OpenCV is heavily used for image processing and provides many methods of altering, inspecting and adding to images. In the C++ interface images are stored in a matrix object `Mat`. Images can have a single channel, such as with greyscale images, or multiple channels as used for RGB images.

OpenCV is used in this project as it provides algorithms useful for head, eye and hand tracking. OpenCV is also well documented and the author has had previous experience using the library.

3.2 Node.js

Node.js is an open source runtime environment for server side applications. It is built on Google Chrome's JavaScript runtime with the aim of being used to build fast, scalable network applications.

The framework of Node.js is modelled to be event driven. This means that unless there is work to do, the Node server will be sleeping. The server does not need to constantly query for responses.

Many connections can be handled concurrently by the server, however there is no risk of deadlock as Node.js does not use locks. The functions in Node are set up so they do not directly perform I/O, this means the process never blocks.

Node.js is used in this project because it is lightweight, easy to set up and works well with WebSockets. Also, because Node applications are written in JavaScript, the functions on both the client side and the server side match up nicely.

3.3 WebSocket

The WebSocket specification defines an API establishing "socket" connections between a user's web browser and a server. This means that there is a persistent connection between the client and the server and both parties can start sending data at any time. This is known as an interactive communication session. [2]

Like Node, WebSockets are event driven which means there is no need for polling. This significantly reduces the overhead of communication between the client and the server. Also, as WebSocket provides a bidirectional communication channel over a single socket native to the browser, there is very little complexity in setting the connection up and using it.

The WebSocket protocol was standardised by the Internet Engineering Task Force (IETF) in 2011 [3] and is currently being standardised by the World Wide Web Consortium (W3C) [4]. The API is available by default in HTML 5.

For use on the server side, there are a number of modules for Node available that provide WebSocket functionality such as Socket.io, WebSocket-Node and WS. WebSocket-Node [5] is used in this project to allow Node.js to use WebSockets. This Node module provides the desired functions as well as examples and documentation.

3.4 WebRTC

WebRTC is an open source project for browser based real time communication via simple APIs, which was released in 2011 by Google. [6] The API definition is drafted by the W3C and is a work in progress [7]. Due to its unfinished state, it is not yet supported by all web browsers and was only tested during the course of this project on Mozilla Firefox and Google Chrome.

Among other components useful for real time communication, WebRTC provides functions to access a user's camera and microphone and to capture media from

them. Of course, the user is first asked if they want to allow access to their webcam or microphone. WebRTC functions are accessible by default as JavaScript functions with HTML 5 in certain browsers.

WebRTC is used in this project to access the user's webcam and capture the video stream in order to send video frames to the backend to be processed by the computer vision program. WebRTC was chosen due to being readily available and easy to use.

3.5 Google Maps JavaScript API v3

As part of their developers site, Google provides software development tools, technical resources and APIs. The Maps JavaScript API allows developers to embed a Google Map onto a webpage and work with it. The API provides many ways to modify and add to both the function and the aesthetic of the default map.

Developers can request access to an API key through their Google account which is free for a limited amount of requests per day. This key is needed to access the JavaScript file that must be included as a script in the HTML file.

This project uses the Google Maps JavaScript API to load a map in the client's browser and to change the center location and the level of zoom in response to the users head movement and gestures. This API was used as it provides a working out of the box map, which is very important for this project, that many users will be familiar with. It also provides functions to easily modify the map in the ways necessary for this project.

3.6 JQuery

JQuery is a fast, small and feature rich JavaScript library. It is not critical to the project however it makes HTML document navigation, document modification and client side scripting much simpler. It is mainly used in this project for accessing elements in HTML documents by their ID.

3.7 HTML 5 Canvas

The canvas tag for HTML was introduced by Apple in 2004 and is a drawable region in HTML code. The JavaScript API is available by default in HTML 5 and provides a full set of drawing functions for graphics and images [8]. The tag and API are used in this project for drawing an image from the video stream coming from a user's webcam to the canvas. This is useful as the canvas object has a function to retrieve the base 64 encoded string for what is currently drawn to it.

4 Computer Vision Techniques

In this section the different computer vision techniques used in this project will be explained.

4.1 Haar Classifiers

Haar-like features are digital image features that are used for object recognition. They get their name from their similarity to Haar wavelets. The OpenCV implementation of object recognition using Haar-like features is based on the initial proposal of the idea by Viola and Jones in 2001 [9] and the improvements on the concept by Lienhart, Kuranov and Pisarevsky in 2002. [10]

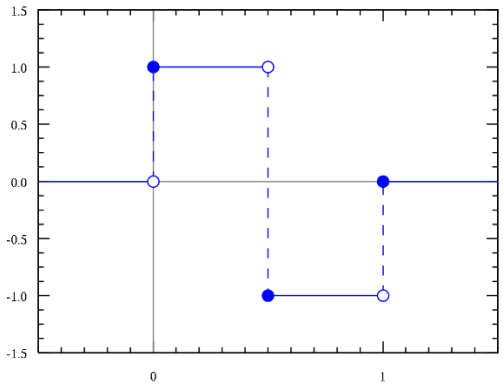


Figure 4.1: Haar Wavelet

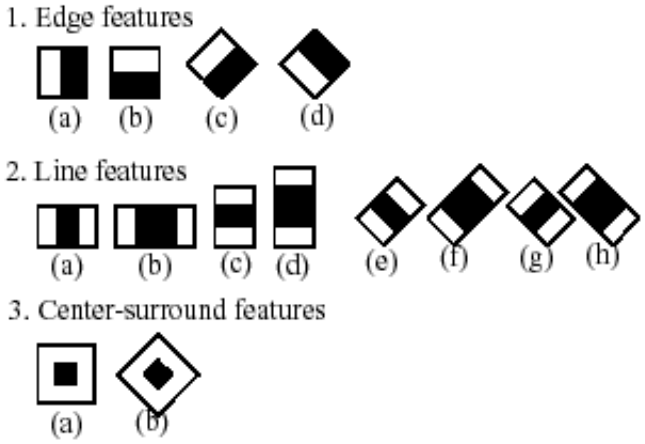


Figure 4.2: Haar-like features used by OpenCV

A Haar Classifier is a cascade of boosted classifiers working with Haar-like features. The word, “cascade”, means that the classifier contains stages. These stages are executed sequentially on a region of interest until all stages are passed or the region of interest is rejected. The word, “boosted”, means that each stage is built out of basic (weak) classifiers that are combined using a boosting algorithm called AdaBoost. [11]

To build a classifier to recognise an object, it must be trained with a few hundred sample views of the object that are all scaled to the same size. These are called positive examples. The classifier must also be trained with negative examples which can be arbitrary images. These negative examples must be at the same size as the positive examples. [12]

Haar-like features are the input to the basic classifiers and at the lowest level return a result based on the difference between the sums of the pixel values in the white regions and the black regions. This works as, after training, the classifiers can expect certain regions of an object to be darker or brighter than adjacent regions. In Figure 4.3 we expect the nose region to be brighter than the eye region.

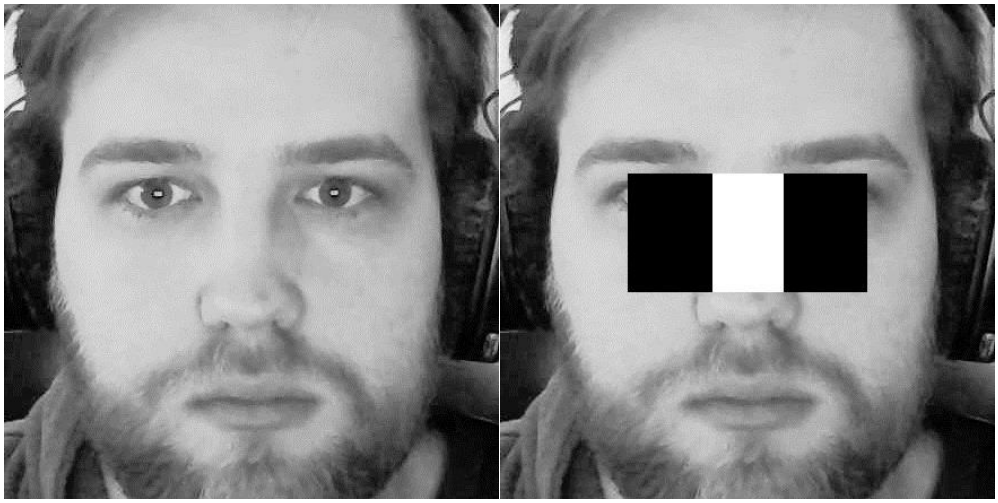


Figure 4.3: Haar-like feature applied to an image

In OpenCV, when a cascade classifier is applied to an image, it checks the entire image for regions that pass the each stage. If a region fails a step, subsequent stages are not applied to that region. To deal with different sized target objects, the classifier can be rescaled and reapplied to the image. This is done numerous

times in order to complete a full search of the provided image for the target object.

4.2 Colour Segmentation/Thresholding

Colour Segmentation is the process of filtering an image based on certain values or ranges of values for the image's colour model. Colour models are by default RGB, however the HSV and YCbCr colour spaces are often used. These models separate the colour and intensity from the lightness of the image, which makes it easier to get consistent results across different inputs.

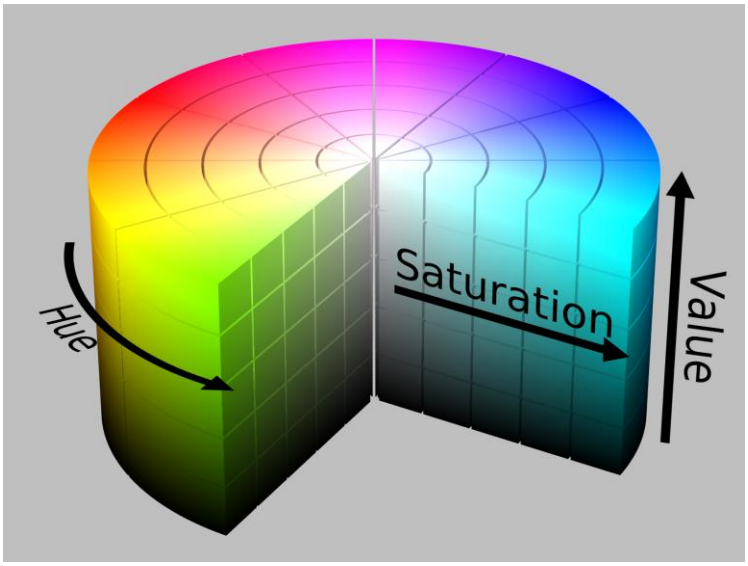


Figure 4.4: HSV Colour Model

The process works by checking each pixel in the image against the specified constraints and setting the value to be 255 (white) if it meets the requirements and to be 0 (black) otherwise. This results in a binary image, where every pixel is either black or white.

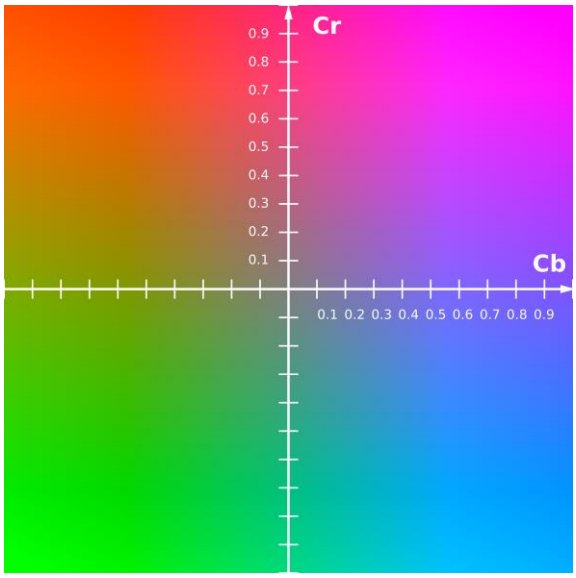


Figure 4.5: 2D view of YCbCr colour model, the missing dimension is the Y (lightness) dimension

Colour Segmentation could also be referred to as Colour Thresholding, however in this report the term thresholding is used in the context of a single channel image rather than images with multiply channels. Both greyscale thresholding and colour thresholding produce binary images.

4.3 Back Projection

Back Projection is a way of recording how well the pixels of a given image fit the distribution of pixels in a histogram model. It is used in this project as a means of feature detection.

Firstly the histogram is created using a single channel from a source image. The source image will be of the target feature, such as skin. In this project the Hue channel from the HSV image is used. Each pixel value has an associated place on the histogram where it is given a value based on its presence in the source image.

Back Projection is then applied, using the histogram, to an image to be searched for the feature. The histogram only works for the channel with which it was created. Back Projection works by going through each pixel in the image, finding the location for the pixel's value in the histogram and storing the value found at that point in histogram at the pixels location in a new output image. [13]

This will create a greyscale image with each pixel corresponding to the input image but changed to have a value between 0 and 255 based on the histogram values. The resulting image is a probability image. The higher the value a pixel has the more likely it matches the histogram.

Back projection is often followed by thresholding, in order to turn the greyscale image into a binary one.

4.4 Morphological Operations

Mathematical morphology is a technique for the analysis and processing of shapes. In the context of OpenCV and image processing, morphological operations apply a structuring element to an input image in order to generate an output image. A structuring element is simply a shape used to interact with an image. It has the purpose of finding out how well it fits the shapes in the target image. Morphological operations are useful to remove noise from a binary image.

The morphology operations used in this project are erode and dilate. When applied to binary images, erode causes the white regions to shrink whereas dilate causes the white regions to expand. These can be used in combination to make an opening operation, which is erode then dilate, or a closing operation, which is dilate then erode.

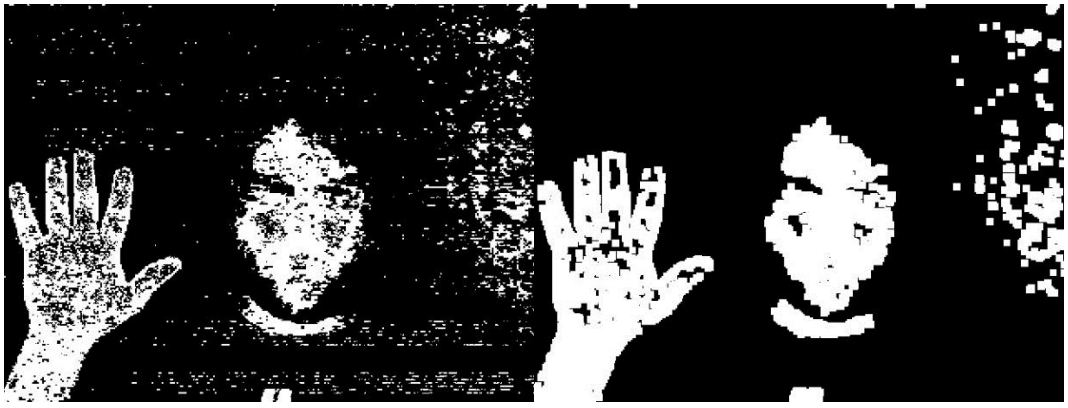


Figure 4.6: Before (left) and after (right) for an erode followed by a dilate

4.5 Connected Components

Connected Components is used in binary images to find regions that are connected. It works by analysing each pixel in the image and giving it a label.

The label is based on the adjacent pixels. There are two ways of commonly looking at adjacency: 4-adjacency and 8-adjacency. These methods are generally used in combination. If there is an adjacent pixel of the same value that already has a label, then the current pixel is assigned the same label. Otherwise the current pixel is given a new label. If two labelled regions are found to be connected, then their labels are made equivalent. [14]

In OpenCV, connected components analysis is done with contour following techniques instead of labelling entire regions. This works in a similar manner but only looks at the boundary points between the binary regions rather than identifying every point within a region. The two methods are essentially equivalent.

5 Implementation

This section will discuss how the web application works, the architecture of the project and the methods in which the project was implemented.

5.1 Architecture

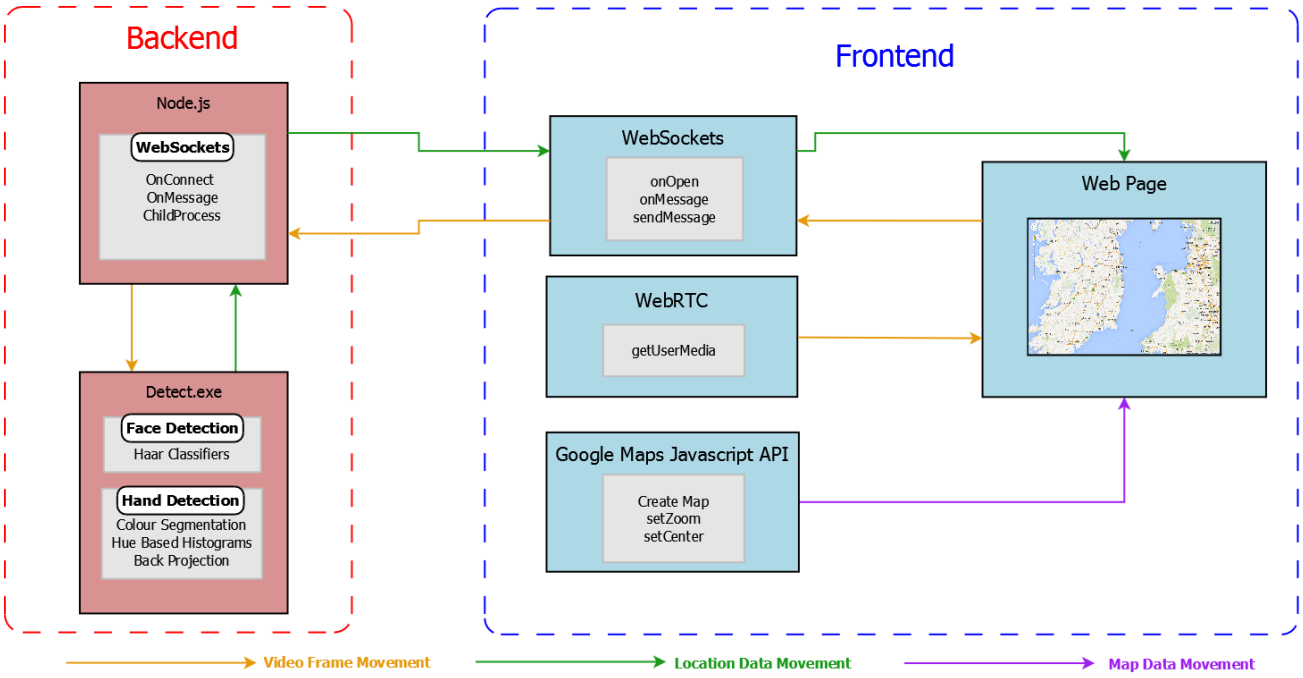


Figure 5.1: Project Architecture

5.2 Frontend

The frontend consists of a HTML file linked to a number of JavaScript files. The JavaScript files consist of JQuery files, and Google Maps API files and three custom files: map.js, webrtc.js and websocket.js.

The HTML body contains two divs and a canvas. One div is used for the initial display. It shows the video stream from the user's webcam and a button to click when they are sitting comfortably. This button links to a `switchToMap` function in the map.js file. The other div is used as the target for loading the map into.

Finally the canvas is used to hold the frame that is going to be sent to the Node server.

5.2.1 Map.js

This file contains a global variable `map`, which is initially given the value `null`, and two functions `switchToMap` and `initialise`. The `switchToMap` function hides the video div and calls the `initialise` function.

The `initialise` function loads a Google Map on the page centred at latitude 53 and longitude -6 at zoom level 8. Google Maps have twenty two levels of zoom ranging from 0 to 21, with 0 being the most zoomed out. Once loaded, the map object is stored as the `map` variable and can be modified using this variable.

5.2.2 Webrtc.js

This file sets up the video stream from the user's webcam. It first checks if the user's browser supports the WebRTC API. If it does not, an alert will inform the user of this. The stream is then set up and sent to the video tag. When this file is loaded by the user's browser they will be prompted to allow access to their camera.

5.2.3 Websocket.js

This file contains a function `convertToBlob`, a number of variables for keeping track of the head location and a variable `ws` for storing the WebSocket object. The function `convertToBlob` takes in a base 64 encoded image and coverts it to binary which is stored as a JavaScript Blob [15] object which are immutable file like objects that represent raw data.

When this file is loaded by the webpage it does three things: opens a WebSocket connection, sets up a timer for sending messages over the WebSocket connection and sets up an event listener for receiving messages over the WebSocket connection. Setting up the connection simply involves specifying the IP address (or domain name) and port number the Node server is running on similar to the following: ws://127.0.0.1:8089.

The timer is set up to run on an interval. Every time it fires the current frame is taken from the video stream. This frame is drawn to the canvas tag which is hidden and so doesn't appear on the user's screen. The base 64 encoded string, for the image that is drawn to the canvas, can then be accessed. This string is then sent to the `convertToBlob` function. The returned value from this function is sent to the Node.js server. The conversion could be done on either the client side or the server side but it is probably best to do as much on the client side as possible before sending the data to the server.

The event listener waits to receive messages over the WebSocket connection. The messages it expects to receive are in the form of the location data for the head, eyes and hand from the sent frame, as well as how far the head is from the camera. These values will be comma separated. When a message is received, it is split on "," and the values placed into an array.

The location data from the sent frame is compared against the default resting values. If any of the values exceed a threshold difference from the resting values, the map is moved in the appropriate direction or zoomed. The individual checks will only result in the map moving up, down, left or right. However the different axes are not exclusive and so two directions can be used at once e.g. down and left.

Changes to the map are made through the following functions associated with the object: `map.setCenter` and `map.setZoom`.

5.3 Node Server

The Node application starts by setting up a http server listening on a specific address and port, which will be the same as the one websocket.js connects to.

The application then sets up the WebSocket server using the http server. It is set to allow a larger than default message size, as it will be receiving image data. The application keeps track of current connections through a list of connected clients, which is added to with each new connection and deleted from upon connection close.

When a new connection comes in, event listeners are set up for that connection to listen for received messages and closing the connection.

When a message containing an image is received, the image is given a random temporary name and it is saved as a file. The compiled C++ code is then called as a child process using the temp file location and an output file location as parameters.

The location data is output by the program through stdout. This is picked up by the Node server and sent as a message to the client. When the child process ends, the temporary files are removed.

5.4 OpenCV Program

The C++ program contains three functions: `detectAndSave`, which does the head and eye detection, `handDetect`, which does hand detection through colour segmentation and `histAndBackProj`, which does hand detection using back projection and the found face from `detectAndSave`.

There are global variables for the file locations of the Haar classifiers, the `CascadeClassifier` objects, a random number generator object, the number of bins for the histogram and a matrix for storing the face area once it has been found.

The program takes two parameters: the location of the input frame and a location to store an output image. The main function first loads in the frame and the Haar classifiers and then calls the functions in succession. Each function takes the input frame as a parameter. When all of the functions have finished and their respective objects are found, the program outputs the location data for each object in a comma separated string.

The vision techniques discussed in this section are explained in chapter 4 of this report.

5.4.1 Head and Eye Detection

This section of the program starts by changing the input frame to greyscale. This means that instead of using three channels such as RGB, the image only has one channel. This channel can be referred to as lightness or brightness as each pixel has one value between 0 and 255 with 0 being black and 255 being white.

The loaded cascade for the face is then applied to the greyscale image. This will attempt to find all of the faces in the image and returns a vector of bounding rectangles for each face. However for this application we only want one face, so we take the largest rectangle. This is based on the assumption that if there are other people in the frame the user will be the one closest to the camera. This will also get rid of any false positives.

Once the face has been found, the program copies the area inside the bounding box from the grey image to a different matrix. The cascade trained for finding eyes is then applied to this region of interest. Similar to the face, a vector of rectangles is returned, of which we only want two. If any are overlapping, they are combined and outliers are removed until there are only two rectangles remaining. These are used as the positions of the user's eyes.

Finding the eye locations allows the user to make more subtle head movements that can translate to map movement. Instead of moving their head entirely in a direction, they can simply rotate their head to look towards where they want the map to move.

Haar classifiers are individually computationally efficient. However as a cascade of classifiers is used and is applied numerous times at different scales to the entire image, the computation can be slow in relation to other, less accurate methods.

To deal with this issue, once the face and eyes have been found initially using the cascade, feature data is recorded which can then be used in feature detection to find the face faster. If and when the feature detection fails the cascade can be applied again to re find the face.

However this program was initially designed to be context free due to the nature of the project architecture. It was made to find the face based on just the input frame. It may be possible to store the feature data for each connected user from the Node.js application. In which case, this method of using the cascade in conjunction with feature detection could work.

5.4.2 Hand Detection through Colour Segmentation

Colour segmentation works by filtering out the pixels in an image based on specific values. For the purpose of finding skin pixels we want to find pixels within a range of values in the HSV colour model. HSV stands for Hue-Saturation-Value. The Hue channel controls the colour, the saturation controls the intensity of the colour and the value is the lightness channel. HSV is a useful model for colour segmentation as lightness is separated from the colour, which is not the case in the default RGB colour model.

The function starts off by converting the input frame to HSV. The `cv::inRange` function is used on this image to extract the ranges that are desired. The connected components are then found in the resulting binary image. The ideal result is two contours: one for the found hand, and one for the face. Any contours with a size too small can be eliminated and any overlapping with the face can be removed too. This should hopefully leave one contour representing the location of the hand.

This method works well if we can guarantee the same lighting each time. However lighting changes and distance from the camera can seriously affect how well this method works and can result in false positives.

5.4.3 Hand Detection through Back Projection

This function is only called if a face has already been found. It takes advantage of the found face by using that region from the input image as a basis for finding a hand.

First a histogram is created using the hue channel from the face area image. Back Projection is performed on the hue channel of the input frame using this histogram. This returns a greyscale “probability” image which is thresholded so that pixels with values over a certain point are set to 255 and all others are set to 0.

This gives us a binary image, but there is likely to be some amount of noise. In order to clear this up, morphological operations are applied to the image. Specifically the image is eroded to remove the noise and then dilated to restore

the other objects. This returns a binary image with less noise which we can use to find the connected components. Similar to the colour segmentation method, the contours are reduced so that we are just left with the hand.

6 Evaluation

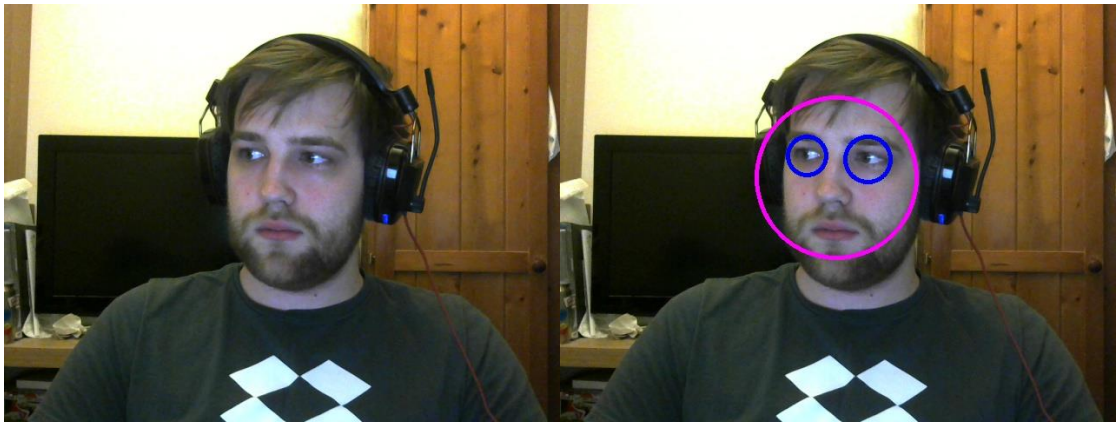
This section will discuss the performance of the application built during this project. There are a subset of test images shown below, the full set will be available on the accompanying DVD.

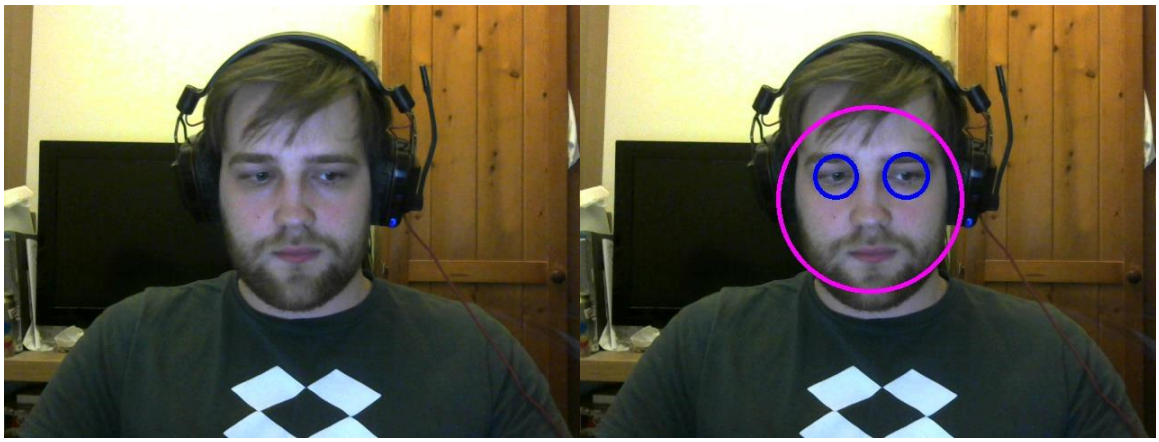
6.1 Computer Vision Processes

The face and eye tracking performs as desired and the application works well with this process. The hand tracking works as intended in certain circumstances but can fail to find a hand or find false positives. It should be noted that the colour segmentation is set up for the lighting used during development and testing, and has proved to not handle lighting changes well.

6.1.1 Face and eye testing

As shown below, the face and eye tracking have high success rates. The face detection only fails if the face is obscured or turned to an angle so extreme it would not be used during interaction with the application. Unfortunately when wearing glasses, the reflection on the lenses from the screen can disrupt the eye detection.





6.1.2 Hand Detection Testing – Colour Segmentation

This process performs well in a controlled environment but occasionally gives false positives. Overlapping bounding boxes can't just be combined into one in the context of finding hands as it is possible that doing so will heavily distort the location data for the hand.

6.1.3 Hand Detection Testing – Histogram and Back Projection

This process performs slightly better than the colour segmentation method. It deals with lighting changes much better but is also prone to false positives.

6.2 Response Time

Testing showed the average time delay between making an action and the map responding to be 0.39 seconds. The majority of this time is spent on computing the Haar cascade.

7 Future Work

This section will discuss options for further development of this project. These ideas were either out of scope for the project or were thought of as a result of working on the project.

7.1 More Gestures

At the moment the hands are only used in the application as an indicator for which direction the map should move. Some simple gestures could be recognised and tied to actions. However gesture recognition beyond the basics is rather difficult, especially when already dealing with unknown lighting and backgrounds. The gestures should also tie to actions that feel natural to the user. This can be problematic as the gestures that feel natural for the desired action can be hard to recognise.

7.2 Other Applications

It would be interesting to see how well the vision side of this project works when used with other applications. It should fit nicely into applications that present a 3 Dimensional environment as the head movements could be tied to moving the camera around the environment. However it is likely the system would have to be paired with some other input to control actions.

7.3 Feature Detection

The Haar classifier method for finding faces could be used in conjunction with feature detection. The process of running a cascade classifier on an image can take significantly longer relative to other vision processes. To speed up computation time, feature detection can be used to find the face and eyes after the initial cascade run. For this to work, some data must be stored by the application. As the pipeline is currently set up, this would have to be done at the server level.

7.4 Eye Movement Tracking

The ability to track eye movement could greatly increase the precision of the application and reduce necessary movement for the user. However accurate eye movement tracking isn't really a possibility with a standard webcam and implementations of this kind of technology tend to take advantage of high quality cameras positioned close to the user's eye.

7.5 Move from server side to client side.

It is possible to remove the dependence on a server doing the computer vision processing by having the program be done by a client side Java applet or something similar. This would significantly reduce the network load of the application but increase the processing load of the client's machine. The user would also have to have the OpenCV libraries installed.

8 Conclusion

The aim of this project was to create a user interface that is controlled through a webcam instead of through more traditional means, such as a mouse and keyboard. This was achieved and the application that was made shows that we can interact with computers using computer vision.

However the project does not require any complex actions to be taken by the user, as the application that was built does not require it. Using a computer vision based interface in other contexts may require a wider range of actions. This is where just using vision as the interface may not be enough.

From this project it appears as though computer vision is limited in what it can do with current interfaces for two reasons. The first, is that interfaces are built with a mouse and keyboard in mind and so can work in ways that sometimes seem unnatural. Yet we, as users, have grown used to them. The second is that when you replace a mouse with a person, the person becomes the controller. This means that the interface will react to everything the person does whether they intend it as input or not.

The second issue can be dealt with through the use of other input, such as voice recognition, wearable tech or through the use of a specific gesture to indicate when to start or stop recording input.

The first issue is harder to solve. Computer vision may not see use in the context of user interfaces unless major changes occur in how we build user interfaces. Vision can be used very effectively when dealing with 3D spaces. The future of interfaces is very vague. New technologies will come along and mature, which will cause users, developers and computers to change in reaction to them. However if user interfaces evolve from a 2D screen to something three dimensional, computer vision should play a big part in how we control them.

9 Attached Electronic Resources

The attached DVD contains the code created during this project and a video demonstration of the application. The disc also contains three folders containing test images for each of the main vision functions. Each file input is stored with a random name and a .jpeg extension and its associated output is stored with the same name and a .jpg extension.

The code was developed on Windows 7 using Visual Studio 2013, but has also been compiled and run on a Ubuntu system. The requirements for running this project are:

- OpenCV
- Node.js
- Websocket-Node module for Node.js

To run the C++ file must first be compiled. This can be done with Visual Studio or with g++ using: “g++ `pkg-config opencv --cflags --libs` main.cpp”.

When compiled the command `var` in both of the node.js files must be changed to point to the produced binary. The default command points to a.out in the same directory. The Node applications can be run using “node filename”.

Each HTML file has an associated Node application: index.html (location-return), which runs the main application and video.html (image-return), which just returns the output image, from the C++ program, to the browser.

When the Node appropriate application is running open the desired HTML file in Firefox.

The code folder also contains a ReadMe that reiterates the information here.

10 References

[1] J. C. Lee, “WiiMote Projects,” 2008.

Available at: http://johnnylee.net/projects/wii/

[2] Mozilla Developer Network, “WebSockets Documentation,” 2015.

Available at: https://developer.mozilla.org/en/docs/WebSockets

[3] I. Fette and I. Melnikox, “The WebSocket Protocol,” Internet Engineering Task Force, 2011.

Available at: http://tools.ietf.org/html/rfc6455

[4] I. Hickson, “The WebSocket API,” World Wide Web Consortium, 2014.

Available at: http://dev.w3.org/html5/websockets/

[5] theturtle32, “WebSocket-Node Documentation,” 2015.

Available at: https://github.com/theturtle32/WebSocket-Node/tree/master/docs

[6] H. Alvestrand, “Google release of WebRTC source code,” 2011.

Available at: http://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html

[7] A. Bergkvist, D. C. Burnett, C. Jennings and A. Narayanan, “WebRTC 1.0: Real-time Communication Between Browsers,” World Wide Web Consortium, 2015.

Available at: http://w3c.github.io/webrtc-pc/

[8] Mozilla Developer Network, “Canvas API Documentation,” 2015.

Available at: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[9] P. Viola and M. J. Jones, “Robust Real Time Face Detection,” 2001.

Available at: http://www.vision.caltech.edu/html-files/EE148-2005-Spring/pprs/viola04ijcv.pdf

[10] R. Lienhart, A. Kuranov and V. Pisarevsky, “Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection,” 2002.

Available at: http://www.multimedia-computing.de/mediawiki//images/5/52/MRL-TR-May02-revised-Dec02.pdf

[11] K. Dawson-Howe, in *A Practical Introduction to Computer Vision with OpenCV*, Dublin, Ireland, John Wiley & Sons Ltd, 2014, pp. 152-158.

[12] OpenCV.org, “Cascade Classification Documentation,” 2015.

Available at: http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html

[13] Intel, “Open Source Computer Vision Library Reference Manual,” Intel Corporation, 2000, pp. ch 10 pg50-51.

Available at: http://www.cs.unc.edu/~stc/FAQs/OpenCV/OpenCVReferenceManual.pdf

[14] K. Dawson-Howe, in *A Practical Introduction to Computer Vision with OpenCV*, Dublin, Ireland, John Wiley & Sons Ltd, 2014, pp. 66-70.

[15] Mozilla Developer Network, “JavaScript Blob Object Documentation,” 2015.

Available at: https://developer.mozilla.org/en/docs/Web/API/Blob

All links were last accessed on 20th April 2015.