

Real-Time Hand Gesture Control of a Virtual Object in Augmented Reality

Final Report

M.L. Williams
18013555

Submitted as partial fulfilment of the requirements of Project EPR402
in the Department of Electrical, Electronic and Computer Engineering
University of Pretoria

November 2022

Study leader: Mr. H. Grobler

Part 1. Preamble

This report describes the work I did in constructing an augmented reality application that allows a user to control a virtual object using hand gestures in real-time.

Project proposal and technical documentation

This main report contains an unaltered copy of the approved Project Proposal (as Part 2 of the report).

Technical documentation appears in Part 4 (Appendix).

All the code that I developed appears as a separate submission on the AMS.

Project history

This project does not build on a previous project. Neural network weights for the hand classifiers were obtained by performing training using TensorFlow. Low-level graphical primitives rendering was performed using OpenGL. Libfreenect was used to interface with and receive depth and RGB camera information from the Microsoft Kinect sensor. OpenCV and Pillow were used to resize and transform images from one colour space to another. NumPy was used for matrix operations as well as optimised array searching. Where other authors' work has been used, it has been cited appropriately, and the rest of the work reported on here, is entirely my own.

Language editing

This document has been language edited by a knowledgeable person. By submitting this document in its present form, I declare that this is the written material that I wish to be examined on.

My language editor was Jason Kamps.



Language editor signature

7 November 2022

Date

Declaration

I, Mitchell Luke Williams understand what plagiarism is and have carefully studied the plagiarism policy of the University. I hereby declare that all the work described in this report is my own, except where explicitly indicated otherwise. Although I may have discussed the design and investigation with my study leader, fellow students or consulted various books, articles or the internet, the design/investigative work is my own. I have mastered the design and I have made all the required calculations in my lab book (and/or they are reflected in this report) to authenticate this. I am not presenting a complete solution of someone else.

Wherever I have used information from other sources, I have given credit by proper and complete referencing of the source material so that it can be clearly discerned what is my own work and what was quoted from other sources. I acknowledge that failure to comply with the instructions regarding referencing will be regarded as plagiarism. If there is any doubt about the authenticity of my work, I am willing to attend an oral ancillary examination/evaluation about the work.

I certify that the Project Proposal appearing as the Introduction section of the report is a verbatim copy of the approved Project Proposal.



7 November 2022

M.L. Williams

Date

TABLE OF CONTENTS

Part 1. Preamble	i
Part 2. Project definition: approved Project Proposal	vii
1. Project description	
2. Technical challenges in this project	
3. Functional analysis	
4. System requirements and specifications	
5. Field conditions	
6. Student tasks	
Part 3. Main Report	xiv
1 Literature study	1
2 Approach	7
3 Design and implementation	9
3.1 Design summary	9
3.2 System pipeline	10
3.3 Hand detection and tracking	10
3.4 Neural network design	15
3.5 Neural network training and testing	22
3.6 Hand gesture recognition	25
3.7 Surface detection	33
3.8 Virtual object implementation and control scheme	38
3.9 Object detection and collision avoidance	42
3.10 Integrated system (main loop)	44
3.11 Multiprocessing	46

3.12 Embedded platform implementation	48
4 Results	50
4.1 Summary of results achieved	50
4.2 Qualification tests	51
5 Discussion	65
5.1 Interpretation of results	65
5.2 Critical evaluation of the design	67
5.3 Design ergonomics	70
5.4 Health, safety and environmental impact	71
5.5 Social and legal impact of the design	72
6 Conclusion	73
6.1 Summary of the work completed	73
6.2 Summary of the observations and findings	73
6.3 Contribution	74
6.4 Future work	75
7 References	76
 Part 4. Appendix: technical documentation	 79
HARDWARE part of the project	80
Record 1. System block diagram	80
Record 2. Systems level description of the design	80
Record 3. Complete circuit diagrams and description	80
Record 4. Hardware acceptance test procedure	80
Record 5. User guide	80
SOFTWARE part of the project	81
Record 6. Software process flow diagrams	81
Record 7. Explanation of software modules	87

Record 8. Complete source code	88
Record 9. Software acceptance test procedure	88
Record 10. Software user guide	88
EXPERIMENTAL DATA	91
Record 11. Experimental data	91

LIST OF ABBREVIATIONS




CNN	Convolutional neural network
FPS	Frames per second
RGB	Red green blue
YCBCR	Luminance blue-difference red-difference
HSV	Hue saturation value
PC	Personal computer
RANSAC	Random sample consensus algorithm
MDL	Minimum description length
GUI	Graphical user interface
2D	Two-dimensional
3D	Three-dimensional
POPI	Protection of personal information act
LTR	Left, towards, right
DFU	Down, forwards, up
DSU	Down, side, up
CPU	Central processing unit
AR	Augmented reality
SBC	Single-board computer
RAM	Random access memory
SSD	Solid state drive

Part 2. Project definition: approved Project Proposal

This section contains the problem identification in the form of the complete approved Project Proposal, unaltered from the final approved version that appears on the AMS.

For use by the Project lecturer	Approved	Revision required
Feedback <div style="text-align: center;">  </div>		

To be completed by the student						
PROJECT PROPOSAL 2022			Project no	HG9	Revision no	1
Title	Surname	Initials	Student no	Study leader (title, initials, surname)		
Mr	Williams	ML	18013555	Mr. H. Grobler		
Project title Real-time hand gesture control of a virtual object in augmented reality						

Language editor name	Language editor signature
Mitchell Williams	
<u>Student declaration</u> I understand what plagiarism is and that I have to complete my project on my own.	<u>Study leader declaration</u> This is a clear and unambiguous description of what is required in this project. Approved for submission (Yes/No)
Student signature	Study leader signature and date
	 2022-06-20

1. Project description

What is your project about? What does your system have to do? What is the problem to be solved?

Augmented reality is a powerful tool to interact with computer interfaces in a natural environment using intuitive human gestures. This project aims to implement a real-time system that can recognize the gestures and positions of a human user's hand and interpret them as different input commands to a virtual object that is instantiated in a live video feed. The system has to recognize the hand gestures and match them against a collection of known gestures and associated virtual object commands. The system must apply the relevant commands to the virtual object in real time with no discernable delay to the user. The virtual object can be handled by the user - moved in multiple directions and rotated in the context of the environment surrounding the user. The virtual object must also interact with the environment it is projected into so that it does not merely float against the background but instead rests on a surface and cannot be pushed through objects in the video stream - like a real 3D object would not be able to be pushed through a solid object.

2. Technical challenges in this project

Describe the technical challenges that are *beyond* those encountered up to the end of third year and in other final year modules.

2.1 Primary *design* challenges

The design of the image processing and hand recognition algorithms will be the principle design challenge of this project in addition to the 3D object generation and image theory required to integrate a virtual object with objects present in a real image. First principles design of a system capable of recognizing the different gestures and positions a hand can make is a primary challenge due to the difficulty of emulating the human visual recognition system. Designing algorithms to integrate a 3D object into a video feed will be another key challenge because of the complexity of three-dimensional environments. The complex nature of video processing, computer vision and three-dimensional scene reconstruction of the environment are design challenges beyond the scope of previous years' work.

2.2 Primary *implementation* challenges

Implementing the hand recognition as well as virtual object manipulation algorithms in real-time on an embedded platform with no discernable delay to the user will be the key implementation challenge. Implementing this system in a real-world setting with the visual occlusions, noise and unpredictable lighting conditions present in a standard office will be a challenge. Inserting even a rudimentary graphical object into the context of a real environment from a video feed will be an implementation challenge due to the unpredictability of those environments and doing this while contending with processing constraints and video transfer speed limitations will be a difficult implementation challenge on the embedded system.

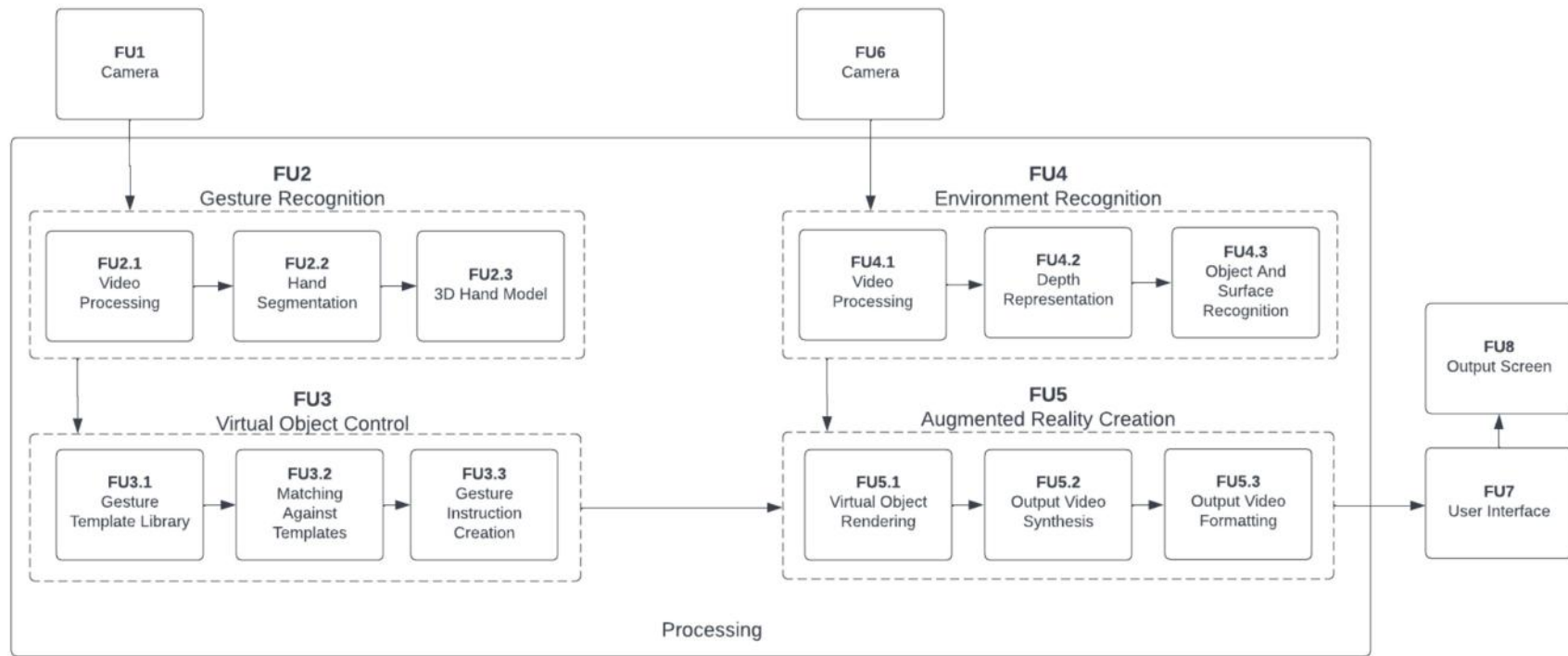
3. Functional analysis

3.1 Functional description

Describe the design in terms of system functions as shown on the functional block diagram in section 3.2. This description should be in *narrative format*.

The user positions themselves in front of a camera (FU1) which captures continuous video of the user and their upheld hand as input to the gesture recognition algorithm (FU2). The video input is first formatted and its individual frames extracted (FU2.1) for the correct input to further processing. Image segmentation (FU2.2) is performed to distinguish the user's hand from the rest of the image. This segmented image is then converted into a 3D model of the user's hand (FU2.3) which is the input to the virtual object control algorithm (FU3). This algorithm first accesses a template library (FU3.1) of existing hand models for different input gestures and then matches the current input hand model to any similar templates (FU3.2) stored in the library - identifying the gesture. This identified gesture is then combined with the parameters of the input model (FU3.3) and sent as a gesture instruction to the augmented reality creation algorithm (FU5). At the same time, another camera (FU6) captures continuous video of the environment in front of the user and inputs it to the environment recognition algorithm (FU4). Here the input video is formatted and has its individual frames extracted (FU4.1) in order to create a depth representation of the environment around the user (FU4.2). This depth information is then used to recognize objects, boundaries and surfaces (FU4.3) that the virtual object can interact with or be affected by. This information is then used in the augmented reality creation algorithm (FU5) where a virtual object is rendered using graphical methods (FU5.1) and based on the gesture and environment input, the two video streams are meshed together (FU5.2) to create a cohesive output video which is formatted (FU5.3) for output to the user interface (FU7). The user interface shows the rendered virtual object in the environment, the current model of the user's hand as well as any error or log messages. The user interface is displayed to the user on a screen (FU8).

3.2 Functional block diagram



4. System requirements and specifications

These are the core requirements of the system or product (the mission-critical requirements) in table format IN ORDER OF IMPORTANCE. Requirement 1 is the most fundamental requirement.

	Requirement 1: the fundamental functional and performance requirement of your project	Requirement 2	Requirement 3
1. Core mission requirements of the system or product. Focus on requirements that are core to solving the engineering problem. These will reflect the solution to the problem.	The system must allow a user to control and manipulate a virtual version of a simple three-dimensional geometric shape object in augmented reality using hand gesture control in real-time.	The system must be able to recognize nine of a user's discrete hand gestures from video input of a single one of the user's hands in real time.	The system must be able to create, render and manipulate a virtual object in a manner consistent with how a real-world version of the same object would be able to be manipulated and controlled.
2. What is the target specification (in measurable terms) to be met in order to achieve this requirement?	A virtual 3D cube that is integrated into the video feed of the environment must be able to be manipulated by a user's hand gestures as well as interact realistically with the environment all with a latency of less than 41.6ms. (24fps)	The system must be able to correctly identify the 9 discrete static hand gestures needed for interacting with the virtual object with a latency of less than 41.6ms from a camera input.	A 20cm-sided three-dimensional cube must be rendered, and then the user must be able to rotate it 360 degrees in the x, y and z-plane, move it up and down, backwards and forwards as well as side-to-side by 10cm with gesture input.
3. Motivation: how or why will meeting the specification given in point 2 above solve the problem? (Motivate the specific target specification selected)	The illusion of augmented reality is created if the object is controlled and rendered at a frame rate of 24 fps - the lowest refresh rate possible before the human eye and brain begins to notice lag and not perceive fluid motion.	These gestures represent the up, down, left, right, backwards, forwards and rotate gestures and being able to identify them in less than 41.6ms allows the system to recognize a user's intended gesture control in apparent real-time (24fps).	These actions represent all the ways an object can be moved in the real-world and thus a cube that can be moved in a virtual setting in these directions and orientations behaves similarly to a real-world cube and creates "augmented reality."
4. How will you demonstrate at the examination that this requirement (point 1 above) and specification (point 2 above) has been met?	Hand gestures will be performed in order to manipulate the object in a desired manner and the corresponding virtual object movement as well as environment's response will be demonstrated as well as the latency taken to perform this action.	A live representation of the user's hand will be displayed in the user interface and 9 gestures will be performed and their correctly interpreted gestures demonstrated, along with the time it took for the system to interpret the gesture.	The cube will be rotated 360 degrees in each planar direction, and then moved sequentially 10cm in every direction on the surface it is resting on to demonstrate the manipulation and rendering of the virtual object.
5. Your own design contribution: what are the aspects that you will design and implement yourself to meet the requirement in point 2? If none, remove this requirement.	The virtual object control algorithm, hand segmentation, hand model, gesture interpretation algorithm, user interface, environmental depth representation, object collision avoidance and surface detection will be designed from first principles.	The hand segmentation and feature extraction algorithms as well as construction of the hand model will be implemented from first principles.	The virtual object manipulation algorithm which includes scaling, translating, placing and rotating the object based on gesture input will be implemented from first principles.
6. What are the aspects to be taken off the shelf to meet this requirement? If none, indicate "none"	Input cameras, an embedded platform, display, image capture, image to array conversion and image display libraries as well as the graphical rendering of the virtual object will be taken off-the-shelf.	A camera, embedded platform and display will be taken off-the-shelf. Additionally, image processing libraries to format and convert the video frames to pixel arrays will be taken off-the-shelf.	The graphical rendering of the virtual object will be taken off-the-shelf with a library.

System requirements and specifications page 2

	Requirement 4	Requirement 5	Requirement 6
1. Core mission requirements of the system or product. Focus on requirements that are core to solving the engineering problem. These will reflect the solution to the problem.	The system must function in user-apparent real time with no visible latency to the user.	The virtual object must interact with the real environment it is rendered inside of in a physically realistic and consistent manner.	
2. What is the target specification (in <i>measurable</i> terms) to be met in order to achieve this requirement?	The virtual object's position and orientation as well as the model of the user's hand must be updated 24 times a second - creating a 24fps video.	The virtual object must remain static for more than 10s on any flat surface in the environment if no input gestures are provided to it and only be moveable 10cm around or over objects in the environment - not through them.	
3. Motivation: <i>how or why</i> will meeting the specification given in point 2 above <i>solve the problem?</i> (Motivate the <i>specific</i> target specification selected)	Human eyes interpret a video that has a frame rate of less than 24fps as choppy and disjointed thus refreshing the virtual object and model of the hands at 24fps creates a smooth and seemingly real-time representation of these objects.	By preventing the virtual object from moving through other objects and surfaces the sense of realism is upheld and the virtual object appears to behave the same way an object in the real world would.	
4. How will you demonstrate at the examination that this requirement (point 1 above) and specification (point 2 above) has been met?	A frame counter will be implemented in the user interface to display how many times per second the virtual object and model of the hand are updated - if the number the frame counter displays is 24 or higher the system will appear to run in real time.	The virtual object will be rendered onto a flat table and a physical 20cm-sided cube will be placed onto the table too. The virtual object will be attempted to be moved through the real cube and the resulting behaviour and distances demonstrated.	
5. Your own design contribution: what are the aspects that <i>you will design and implement yourself</i> to meet the requirement in point 2? If none, <i>remove this requirement.</i>	The frame counter, user interface, model of the hand and virtual object manipulation algorithm will be implemented from first principles.	The creation of a depth-representation of the environment, collision avoidance as well as boundary detection algorithms will be implemented from first principles in addition to the virtual object manipulation algorithm.	
6. What are the aspects to be taken off the shelf to meet this requirement? If none, indicate "none"	The graphical rendering of the virtual object as well as input cameras will be taken off-the-shelf.	An input camera, embedded platform and display will be taken off-the-shelf.	

5. Field conditions

These are the REAL WORLD CONDITIONS under which your project has to work and has to be demonstrated.

	Field condition 1	Field condition 2	Field condition 3
Field condition requirement. In which field conditions does the system have to operate? Indicate the one, two or three most important field conditions.	The system must function in standard indoor lighting conditions for computer work	The virtual object will be manipulated using a single visible hand at a time.	The virtual object must be able to be manipulated when it is placed in the user's immediate vicinity.
Field condition specification. What is the specification (in measurable terms) for this field condition?	The room must be at 300lux brightness or greater.	A maximum of 1 hand at a time can be present in the camera feed and a maximum of 1 finger can be partially occluded by the hand it is attached to.	The virtual object must be placed within a 2m radius of the user and not more than 2m away from the camera.

6. Student tasks

6.1 Design and implementation tasks

List your primary design and implementation tasks in bullet list format (5-10 bullets). These are *not* product requirements, but *your* tasks.

- Research will be conducted on how gesture recognition is performed using hand models and the calculations involved.
- Two cameras and a depth-sensing device must be interfaced with a PC to enable communication and image transferral.
- Image processing algorithms for compressing and formatting video input must be implemented.
- A hand image segmentation algorithm must be designed and implemented to separate the pixels of a hand from the background of a camera image.
- A hand model creation algorithm must be designed to create an accurate model of a hand and its current gesture.
- A gesture template matching algorithm must be implemented to match an input gesture to an assembled template library of gestures.
- A virtual object rendering algorithm must be developed to create a virtual object and insert it correctly into a live video feed.
- It must be ensured that all algorithms and processing is performed in real-time with no discernable delay to the user.

6.2 New knowledge to be acquired

Describe what the theoretical foundation to the project is, and which new knowledge you will acquire (*beyond* that covered in any other undergraduate modules).

- Understanding of computer vision techniques such as image segmentation and feature extraction will be developed.
- Knowledge of statistical methods for confirming the accuracy of machine learning models will be developed.
- The techniques of artificial intelligence and deep learning will be researched and used to classify hand image data.
- The background of real-time video compression, image processing and analysis will be required in order to handle the large amounts of input video data required.
- An understanding of computer graphics, visual interfaces and three-dimensional digital representations of objects will be developed.
- The mathematics behind computer vision and image processing will be researched and utilized in order to synthesis the output video.

Part 3. Main Report

1. Literature study

With the increase in the proliferation of powerful personal computing hardware it has become feasible to create augmented reality applications that integrate virtual objects with a user's physical environment. Similarly, modern computer systems can perform real-time inference on a large range of alternative inputs and return useful results – this has led to the advent of human-control inputs to computers like hand gesture control using regular webcams. These two fields - augmented reality and gesture control, can be combined to give a user a natural and intuitive control mechanism for interactive and visual applications.

The literature is studded with examples of applications that use this combination of technologies, such as Billingham [1] who utilises a Microsoft Kinect depth and Red-Green-Blue (RGB) camera to treat agoraphobia by creating virtual spiders that the user can interact with using their hands in an augmented reality application. This is accomplished by extracting point cloud depth and RGB camera data of a table and the user's hands and segmenting the hands out from the background using the point cloud depth information from the Kinect sensor. The virtual spiders are then overlaid on the user's hands in software and displayed on a desktop monitor - creating the illusion of augmented reality. The same researchers also utilise the Kinect sensor to reconstruct virtual cars on a tabletop using surface reconstruction and enable interactions between those virtual cars and real-world objects by use of a virtual mesh that is updated in real-time as the real-world objects are moved around.

Collisions between the virtual cars and real-world objects are detected by checking if the hands and virtual mesh are in the same position or have the same depth data values. Additionally, the system locates the user's hands by segmenting the RGB image from the Kinect sensor by skin color and a curve-finding algorithm is used to locate the fingertips - a rudimentary but effective solution to hand tracking if the background environment is sufficiently devoid of noise.

Baldauf [2] uses gesture input from a mobile phone camera to select, shrink and zoom in on virtual objects presented in the environment as well as to recognise gesture volume controls for a music application. The system makes use of a skin detection algorithm to create a binary image representing the hand which is then de-noised and supplied to an OpenCV algorithm to find the hand's contours. From these contours the palm of the hand is found by finding the largest circle that can fit inside of the segmented hand. Following this, a distance algorithm is used that finds curves a certain distance away from the palm to detect the fingertips of the hand. The location of the fingertips and the distance between the index finger and thumb is then used to either grow or shrink a virtual cube superimposed on the image or increase and decrease the volume of a music player application - real-world use cases for applications that rely on gesture input.

The ability to locate virtual objects in the context of the real-world environment in an augmented reality application is important if realistic interaction is to take place. Kato [3] implements a tabletop augmented reality application for handling small virtual shapes and cards that relies upon a global coordinate system and paper tracking fiducials placed on the tabletop to give both virtual objects and real-world objects their coordinates in the global coordinate system and then be able to control virtual object movement and behavior accordingly. The

global coordinate system is defined relative to the paper fiducials placed on the table and the camera's position is determined from triangulating its distance to each fiducial. Furthermore, the system allows for rotation and tilting of the virtual shapes by comparing current global coordinates of a virtual shape to previous coordinates and if the difference is great enough rotating or tilting the shape in the correct direction.

Similarly, Buchmann [4] uses a world coordinate system in an urban planning augmented reality application that tracks the position of virtual objects as well as the user's hand and current gesture to determine if an object should be grasped, moved or released at any given time. This also allows for collision avoidance as the same coordinate system is shared by all objects – real or virtual. The system also relies upon paper fiducial markers placed on the tabletop and on a glove that the user dons for input to the world coordinate system. A real-time model of the user's hand and the current gesture is created from the orientation of the fiducial markers in relation to a camera.

Since the fiducial markers on the glove and tabletop all share the same world coordinate system they can be easily compared and allows the user to interact seamlessly with the virtual buildings and roads rendered in the urban planning augmented reality application. The current gesture of the hand is recognised by comparing the distance of the fingertip fiducial markers from each other and by checking if the fingertips are a certain distance inside or away from virtual objects' positions. Thus simple gestures such as grabbing, releasing and dragging can be recognised. Small electronic buzzers that vibrate when the user touches or drags an object are used to provide the user with haptic feedback when using the application.

Augmented reality applications also sometimes depend upon plane detection to identify surfaces depending on their application. Schnabel [5] utilises the Random Sample Consensus (RANSAC) algorithm to search through unstructured point clouds of depth information and output shapes that demonstrate where planar surfaces exist in the depth data. This works by calculating the normal vectors for randomly selected pixels in the point cloud and growing a region that contains all similar normal vectors until no more similar vectors can be found and then adding that region and its shape to an existing array of known planar surfaces - while removing the pixels from the point cloud data. In this way, planar surfaces can be found efficiently in depth point clouds and virtual reality objects placed on those detected surfaces.

This is also partially the approach taken by Nuernberger [6] where a Microsoft Kinect is used to capture depth data. Following this, the data is filtered using an exponential filter and surface normal vectors are computed at each pixel and then used to detect edges of objects in the depth point cloud. From these edges, the Hough Transform is used to find dominant lines and the RANSAC algorithm to find points inside the edges of those lines. The Hough Transform is another widely-used algorithm for plane detection and works by finding all the planes each point in a point cloud can fall on and then using a data structure called an accumulator to iteratively find the planes that contain the most points in the point cloud. In this way the dominant planes present in the point cloud can be estimated.

The application by Nuernberger [6] goes on to estimate planar surfaces from the extracted edges and project virtual shapes onto the surfaces. Other applications such as by Liu [7] utilise convolutional neural networks to estimate planar surfaces but the industry standard is to use some version of RANSAC for simple applications due to its insensitivity to noise and

overall simplicity, such as by Yang [8] with the combination of RANSAC and the Minimum Description Length (MDL) algorithm to find planes in unstructured point cloud depth data.

All of the applications that use hand gestures as input and hand gesture control itself can be considered as solving the two sequential problems of hand pose estimation and gesture recognition based on the hand pose predicted. Gesture recognition is either performed using the classical approaches described above that centre around hand contour-finding or fiducial marker detection, or as is more common in recent approaches, is performed using machine learning approaches such as support vector machines, Naïve-Bayes classifiers and convolutional neural networks as by Ahmed [9] for the recognition of Indian sign language based on hand coordinate input. It can also be accomplished by extracting features from the input image using Gabor Wavelet Transforms and gradient local-auto correlation and then providing these features to a multi-layer perceptron or K-nearest neighbors system such as by Sadeddine [10] to recognise sign language. The complexity of the algorithm required in gesture recognition depends on the static or dynamic nature as well as diversity of the input gestures.

What is apparent from the literature, however, is that the main challenge of gesture recognition is first acquiring an estimation of the user's hand pose from camera input – solutions to this problem have been proposed and implemented since the 1990s. These early solutions [11] relied on classical approaches to hand pose estimation such as using The Continuously Adaptive Mean Shift algorithm to recognise the very high or low saturation of image pixels in the Hue, Saturation and Value colourspace (HSV) to segment a hand from its background and then using a curvature-based least-squares fitting algorithm for detecting the contours of the hand such as fingertips. Additional information about the hand pose is estimated from the contours of the palm and used to find the convexity defect points between fingertips - these allow the orientation of the hand relative to the camera to be found.

An alternative to hand pose estimation is to use a physical glove with fiducial markers on it as used by Buchmann [4] to detect the position of a user's hand in space. However, with the advent of modern computing power and the rise of deep learning, the literature has been saturated with machine learning approaches to hand pose estimation that require none of the specialised hardware or highly specific algorithms that previous implementations required.

This is visible in a state-of-the-art hand-tracking application created by Google - dubbed Mediapipe Hands [12], which uses a series of convolutional neural networks to train a palm detector and hand landmark model to output coordinates of hand joint landmarks. The system runs in real-time on mobile devices and is trained using real images of hands as well as synthetic hand models. In order to reduce the complexity that the hand joint coordinate neural network must deal with, a palm detector is first implemented using a single-shot detector to predict a bounding box around the palm present in the input image.

Once the palm has been detected, the image is cropped to this bounding box and a convolutional pose machine is used to create a confidence map that shows the probability of finding a fingertip at any given point in the input image. This convolutional pose machine is comprised mainly of simple convolutional and pooling layers. From the generated confidence maps, the system outputs the x,y and relative z coordinates of the 21 hand landmarks which together represent an accurate depiction of the current pose of the user's hand.

Similarly, Qing [13] uses a deep convolutional neural network with just convolutional and pooling layers to output three-dimensional joint locations for a hand based on depth image input. This is why the literature often refers to hand pose estimation as hand joint-regression. The advantage of the very deep convolutional neural network is that it obviates many of the intermediate feature extraction tasks that would otherwise have to be designed by hand and instead allows the system to learn these itself.

The architecture of the neural network used by Qing relies upon eight convolutional layers, four pooling layers and three fully-connected layers at the output of the system. This is aided by batch normalisation and allows the system to take in a simple depth image and regress all the way to coordinates for the various landmarks of the hand. Gomez-Donoso [14] employs a similar architecture to predict joint locations by first using a convolutional neural network to detect and segment the hand using a box prediction system reminiscent of the YOLO9000 architecture [15], and then regress the joints of the hand using a large convolutional neural network based on the RESNET50 architecture [16].

Specifically, the system uses a convolutional neural network to detect the probability of a hand being present in various regions of the image. This is accomplished using an object localisation or box prediction for the hand that uses a smaller version of the full YOLO9000 architecture - nineteen convolutional layers and five maxpooling layers. Once the hand has been detected, the cropped image of the hand is passed to a modified RESNET50 convolutional neural network that is adapted to regress normalised three-dimensional (3D) hand joint coordinates from the simple cropped input image of the hand. The two neural networks are trained using a combination of existing weights and a custom dataset of hand poses taken with a Leap Motion depth and RGB sensor.

Alternatively, there are implementations of hand pose estimation that make exclusive use of depth camera input. This depth input is often modified in an intermediate transformation such as a heatmap to show where each joint of the hand is likely to be and regresses the location of the joints from this intermediate layer. This is the approach taken by Chen [17] where a convolutional neural network regresses joint locations from feature regions which are themselves extracted from feature heatmaps created by depth image input.

Specifically, the system built by Chen takes in a depth image and rough previous hand pose estimation. The depth image is run through a small convolutional neural network (six convolutional layers and two residual connections) that outputs feature heatmaps for each joint of the hand which are used in combination with the previous rough hand pose estimation to extract feature regions for each joint of the hand. These feature regions are then hierarchically connected to a final convolutional neural network which outputs the regressed coordinates for various joints of the hand and represents the hand pose estimation of the entire system.

Ding [18] and Ge [19] also make use of heatmaps of joint coordinates and subsequent fine-tuning algorithms to output joint locations based on the intermediate layers. Ding [18] simplifies segmentation of the hand from the background by assuming that the hand is the closest object to the camera and by using a depth camera, extracts a fixed region of depth and RGB information from around the closest depth value to the camera. This extracted region is then resized and fed into a small convolutional neural network that outputs hand pose parameters which are used with another hand model layer (featuring six convolutional

layers) to regress rough estimates for the hand joint locations. Fine-tuning is then performed to modify the output of this layer by giving larger weight to predictions that match up with the initial rough hand pose estimation and reducing the weight given to predictions made away from the hand's centre as those are more regularly inaccurate. Coupled with many rotations and translations in a data augmentation process, the system is used to accurately provide an estimate of hand poses from RGB and depth image input in real-time.

Ge [19] takes an input depth image and projects it onto three different orthogonal planes - the x-y, y-z and z-x planes of a bounding box around the image. Separate convolutional networks are then used to take the depth projections described above and output feature maps which when combined through a final fully-connected layer, output a heatmap showing the probability of a hand joint being present at each coordinate - giving an accurate hand pose estimation from multiple three-dimensional views.

Taking a different approach, Wu [20] develops an architecture that involves calculating a skeleton-difference loss network to regress the joints of a hand skeleton based on depth camera input. The system works by accepting depth images as input and uses a ZF-Net [21] inspired convolutional neural network to generate bounding boxes for a detected hand in the input image. The image is then cropped to the dimensions of this bounding box and passed to the skeleton-difference loss neural network. This network seeks to minimise the angle between all the joints of the hand skeleton and between the joint length and ground truth joint length. This network predicts the location of the hand joints using a one-hundred-and-one layer recurrent neural network. Additionally, this implementation is developed to be robust to occlusions of the hand by objects held in the hand.

Hand pose estimation itself is a sub-field of full-body pose estimation which is a problem solved by Toshev [22] and which takes advantage of a hierarchical progression of increasingly-fine-grained pose regressors for the joints of a whole body and is comprised at its core of simple convolutional layers stacked after one another.

It is evident that the advent of deep learning has yielded a large number of new approaches to hand pose estimation and that the extensive use of convolutional neural networks is the modern approach most preferred in academia. This is due to the ease of not having to implement detailed representations of low-level hand shapes, patterns and methods of identifying these features in input imagery but rather instead training a deep learning system to identify and learn these low-level abstractions using vast amounts of training data and optimised architectures such as the convolutional neural network.

The majority of hand pose estimation systems surveyed above rely on some form of detection of the user's hand or palm as a provisional step to hand pose estimation. This is because detecting the hand allows a system to crop the input RGB or depth image to only those dimensions that include a hand and allow background noise and interference to be suppressed - increasing the accuracy of hand pose estimation and hand joint localisation systems as well as decreasing the computational complexity and training time needed to train these large neural network systems to be robust against noise and various different inputs. The classical approaches to hand detection and fingertip detection mainly rely on skin color or hue segmentation followed by contour and curvature-based algorithms that can detect the contours and curves of the user's hand and locate these in a segmented input image - however

these have their shortcomings when it comes to noise tolerance and reliability.

Returning to the implementation of augmented reality systems - many augmented reality applications have been created that rely on hand and gesture input and all use some form of global or world coordinate system shared by real-world and virtual objects. This allows the position of these objects to be related to each other and for collisions and interactions between these objects to be modelled and rendered. Classically, paper fiducial markers were used to orient the camera and synchronise virtual as well as real-world objects to the global coordinate system, however modern approaches have dispensed with these as depth-tracking devices such as the Microsoft Kinect and Leap Motion sensors have become available for academic use and can fulfill much the same purpose.

Considering the broad body of literature on hand gesture control of virtual objects and their applicability to augmented reality applications, several design choices have been informed for the system to be implemented. It is evident that the preferred approach in the literature for hand pose estimation and gesture recognition is to use a deep-learning architecture to regress hand joint coordinates from either a depth or standard RGB camera input. Gesture recognition can either be performed by deep-learning approaches or by classical calculations depending on the complexity of the required gestures.

Augmented reality and the combination of virtual reality objects with real-world objects can create immersive and useful applications when a suitable input camera is used and a shared coordinate system is established to track both virtual and real objects and prevent collisions between them. The use of a RANSAC-based algorithm on normal vectors calculated from the depth information of a suitable camera can be used to estimate planar surfaces and place objects on those surfaces in augmented reality or even to aid in collision avoidance between objects in the shared coordinate space.

The use of large-scale neural networks in the literature for gesture recognition and joint regression is mainly used for detecting large numbers of gestures or finding the coordinates of tens of hand joints in an input image. Scaling down the output of these networks can significantly reduce the complexity needed in their construction and in the amount of layers needed for accurate operation. Thus, designing a system that only differentiates between a handful of gestures or localises one or two hand landmarks will be much cheaper to develop computationally and allow better operation using an embedded device and first-principles algorithms.

The additional use of pre-processing using skin color and hue segmentation, depth data as well as hand detection and bounding-box algorithms - whether classical or using a deep-learning approach, will also greatly reduce the complexity of the gesture recognition and hand pose estimation design work to be completed. In conclusion, a system will be developed that can accept user gesture input using a deep-learning approach coupled with targeted pre-processing and then translate that gesture into meaningful instructions for a virtual object present in an augmented reality scene that presents realistic interactions between it and real-world objects and uses a global coordinate system to prevent object collisions.

2. Approach

The problem to be solved is how to create a real-time system that uses hand gesture control to manipulate a virtual object in augmented reality. The original project concept expected the solution to be implemented on a PC but this was later modified to include functioning on an embedded platform. The expectation is thus that the system will run on both platforms but with decreased performance on the embedded platform. The system necessitates the integration of hand gesture recognition, environment recognition, virtual object control and augmented reality creation subsystems. Various design choices were made for each of these components based on the literature reviewed and constraints imposed upon the system.

The creation of an augmented reality application first relies on some form of input which can be modified with virtual elements. This can take the form of an ordinary camera that produces RGB images or a more specialised sensor that provides depth images with values representing the distance of each pixel in an image from the sensor. Both of these sensors are combined in affordable modern systems like the Microsoft Kinect and are useful in that they allow all normal image processing and recognition tasks to be performed on the RGB image but then also provide depth data which can be used for plane estimation, collision avoidance and provide a whole additional axis of information to be used by an augmented reality application. Thus the Microsoft Kinect was chosen as the input sensor for the system.

This depth information was used to find the surface of the table the virtual object was to be placed on, as part of the object and surface recognition subsystem. This was done to allow realistic behaviour of the cube when it is left alone in the system or attempted to be pushed through the table. The depth data was used to calculate the normal vectors for every pixel in the image and to extract the dominant planes in the image using the Random Sample Consensus (RANSAC) algorithm. This allowed the system to find the surface of the table in the image by finding the largest plane at the bottom of the image.

Alternative approaches considered included naive colour segmentation approaches to distinguish the table surface by its colour and deep learning methods to find planes in RGB imagery but these methods would be respectively very sensitive to colour noise in the image and computationally expensive. Additionally they would not take advantage of the depth information that can be used to quickly find planar surfaces and thus the approach described above was chosen.

The object recognition subsystem needed to be able to detect objects around the user and prevent collisions between the virtual object and these real-world objects. The plane detection approach above could be extended to detect perpendicular planes as well but a simple solution was to use the depth information provided by the Kinect sensor to check if an area around the user's hand is the same distance away from the sensor as the hand. In this way, the system can detect if the user's hand holding the virtual object is about to come into contact with an object that is in its way and effectively detect a collision and stop movement of the user's hand and virtual object.

Movement of the virtual object was controlled by gestures of the user's hand and these gestures had to be detected in the gesture recognition subsystem. Classical methods for detecting the current state of the hand include colour segmentation of the human skin hue using the

luminance, blue-difference and red-difference (YCbCr) colour space and contour-finding algorithms to identify the fingertips of the hand. These contour approaches break down however when more complicated gestures like a closed fist or hand with fingers occluded have to be detected. Thus the approach taken was to use a Convolutional neural network (CNN) to take in the Red-Green-Blue (RGB) image from the Kinect sensor and output a classification of the current gesture of the hand. To improve the accuracy of the classification however and reduce the computations necessary to achieve this, the input image was preprocessed using some of the classical approaches including YCbCr colour segmentation of human skin hue and resizing based on where a hand is detected in the input image.

This hand detection preprocessing additionally serves to give the position of the hand in the image at any given time and whether or not the hand is near or far away from the current location of the virtual object. In this way, the object can be set to only be manipulated by the hand when they are very close to each other - as would occur in real life.

The approach taken to detect complicated gestures was not to classify each individual complex gesture but rather to detect a large number of simple classifications and then combine them to form a larger model of the current hand gesture. This resulted in using four different CNNs to detect whether a hand is open or forming a fist, whether the hand is pointing downwards, sideways or upwards, whether the hand is pointing downwards, forwards or upwards and whether a hand is facing to the left, right or towards the camera. The training of these networks allowed further accuracy to be gained by carefully creating and selecting the training data for the classifiers.

Once the current gesture is detected, the virtual object must be manipulated accordingly. This was done by taking in the current gesture and converting it to a movement command for the object based on its previous position and orientation. If the gesture detected now informs a rotation of the object, this is combined with the current position of the hand and a command issued to rotate and translate the object to the correct location in the image.

The augmented reality creation subsystem is the final component of the system and outputs the final rendered video. Various three-dimensional graphics rendering solutions exist but in order to keep the system lightweight and render only a simple cube, the OpenGL framework was chosen due to its flexibility, support to run on both a PC and embedded platform and ability to render graphical primitives like lines, vertices and planar surfaces with minimal code. The underlying OpenGL code was used to render a quad on the screen which displayed the input image from the Microsoft Kinect camera and then render a cube on top of that quad in the various rotations and translations as informed by the user's gesture inputs.

The choice of Python for the system's programming language is due to support for OpenGL [23] and Kinect interfacing [24] wrappers in Python and the expected need for much iterative development of the machine learning classifiers - Python being dynamically typed and an interpreted language lending itself to this use case. The use of linear algebra libraries such as NumPy [25] for certain matrix and mathematical operations was also predicted and this and other libraries used are written in the lower-level C language and thus their use with Python obviates the need to use a lower-level language to achieve greater computational performance.

3. Design and implementation

3.1 Design summary

This section summarises the project design tasks and how they were implemented (see Table 1).

Deliverable or task	Implementation	Completion of deliverable or task, and section in the report
The hand detection, segmentation algorithm and hand model had to be designed and implemented by the student.	The student completed the design and implementation using classical colour approaches.	Completed. Section 3.3.
The gesture interpretation and detection algorithm had to be designed and implemented by the student.	The student completed the design and implementation of the CNNs for gesture recognition from first principles barring some optimisations.	Completed. Section 3.6.
The depth representation of the environment, collision avoidance algorithm and boundary detection system had to be designed and implemented by the student.	The student completed the design and implementation from first principles by making use of the Kinect depth information.	Completed. Section 3.9.
The surface detection algorithm had to be designed and implemented by the student.	The student completed the design and implementation using a modified RANSAC algorithm for plane detection.	Completed. Section 3.7.
The frame counter, user interface and virtual object control algorithm had to be designed and implemented by the student.	The student completed the design and implementation using the OpenGL library.	Completed. Section 3.8.

Table 1.
Design summary.

3.2 System pipeline

The functional block diagram presented in the project proposal outlined various subsystems that the overall system would require in order to function correctly. This, in tandem with the development and implementation process, has lead to the establishment of an overall pipeline that represents the entire final system. It accepts data in from the Microsoft Kinect sensor and processes it in subsystems to perform gesture detection, virtual object manipulation, environment recognition as well as augmented reality creation. The classifiers are performed in their own processes and all the other tasks required to make the system function are processed in another. This improves system speed and performance and is outlined in more detail in Section 3.10. The overall pipeline of the system is presented in Figure 1 and the remainder of Section 3 is dedicated to expanding on the details of each subsystem of the pipeline.

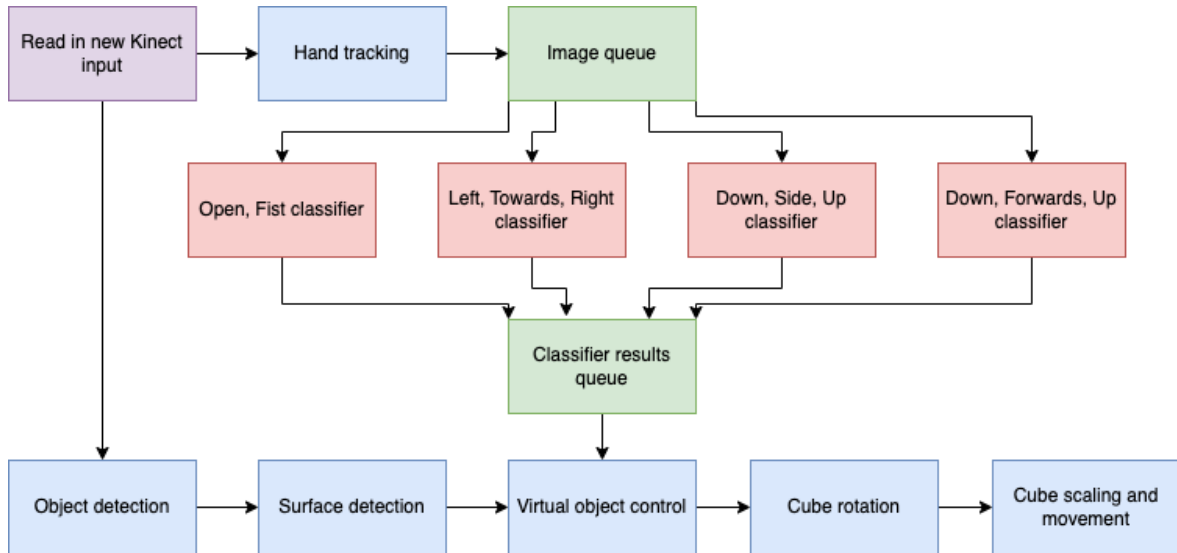


Figure 1.
Overall system pipeline.

3.3 Hand detection and tracking

In order to simplify the classification of hand gestures it was necessary to first locate the hand in the input image frame and crop the image to only those pixels containing the hand. This reduced the complexity of the classifier that needed to be trained and removed a large amount of background noise, interference and features that were not relevant to the current state of the hand. Additionally, by locating the hand in the input image frame, the position of the user's hand relative to the virtual object could be determined. The algorithmic description of the hand detection and tracking algorithm is presented in Algorithm 1.

Algorithm 1 Hand detection and tracking

```

1: resize image to 80 x 60
2: increase saturation by 50%
3: convert image to YCbCr colour space
4: segment image based on skin hue range
5: convert image to binary form
6: sum pixels along all edges
7: find largest edge sum
8: for col = 1, 30, ... do
9:   for row = 1, 23, ... do
10:    calculate sum of pixels in region[row][col]
11:    if pixel sum  $\geq$  1200 then
12:      find delta from image edge entered
13:      append delta to edge_deltas
14:      append pixel sum to region_sums
15:      append row and col to r_region_counters and c_region_counters
16:    end if
17:  end for
18: end for
19: if region_sums contains values then
20:   hand_present = true
21:   set closest_delta_index to index of min delta from entry edge
22:   for col=closest_delta_index column -8, closest_delta_index column +8... do
23:     for row=closest_delta_index row -8, closest_delta_index row +8... do
24:       if region_sum[row][col]  $\geq$  max_sum then
25:         crop_coords = [max_c, max_c+20, max_r, max_r+15]
26:         widen crop_coords by 10 on all sides
27:         new_depth = smallest value in crop_coords
28:       end if
29:       hand_box_coords = [max_c+10, max_r+7.5]
30:       set extracted hand using crop_coords
31:       set normalised extracted hand by dividing by 255
32:     end for
33:   end for
34: end if
35: return new_depth, hand_box_coords, extracted_image, edge_max_index

```

Digital images are stored as sets of numbers representing different concentrations of elemental colours - colour models. Different colour models allow different operations or different visualisations to be observed. The YCbCr colour model is a transformation of the RGB colour model into two colour components and one light intensity component. This effectively creates a grayscale version of the image in the form of the luminance component and two additive components that when combined restore the original colour of the image. The components of the RGB and YCbCr colour model are shown in Figure 2. By producing the two chroma components in the YCbCr model, valuable information becomes available as the hues of the

image and their difference from pure blue and pure red become available.

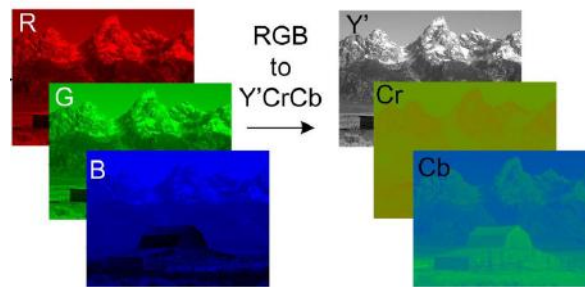


Figure 2.

The components of the RGB and YCbCr colour model. Reproduced from [26].

While people of different races have different skin colours, the hue of human skin is relatively uniform and thus by converting images to a colour model like YCbCr where the chroma components show the deviation of a pixel colour from a specific hue, human skin can be effectively segmented in an image and is the approach taken in the development of the hand detection and tracking subsystem.

In designing the skin colour or rather skin hue segmentation algorithm, the range of values that the skin hue could fall in had to be experimentally determined. There are known values but the effect of fluorescent lighting and the particular white balance of a camera can affect this. Thus a small test program was created to produce a segmented binary image based on the specified range of the hue values allowed for the luma, blue-difference chroma and red-difference chroma components of a YCbCr image. The result is a binary image that displays only skin hues alongside some background noise and is visible in Figure 3.

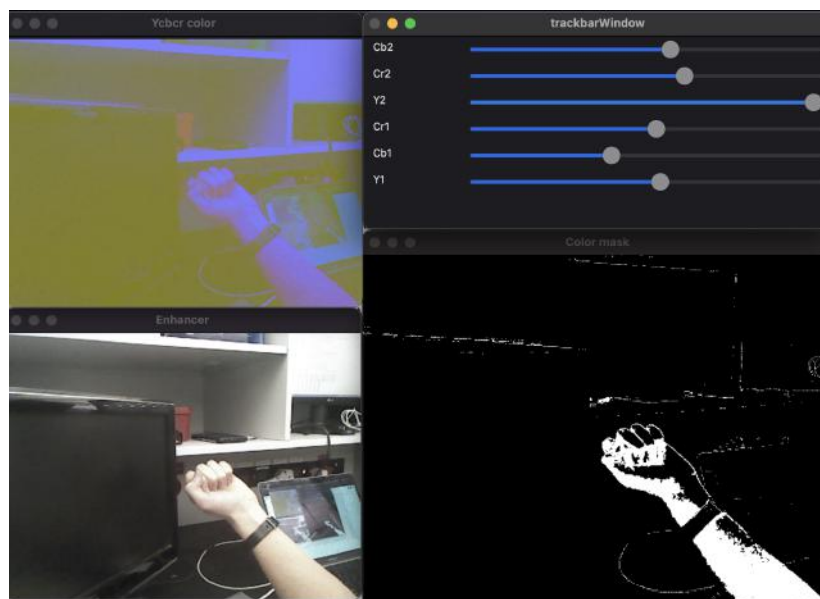


Figure 3.

Test program for varying YCbCr segmentation range.

Increasing the saturation of the image before the YCbCr colour transformation took place increased the differentiation between the hand and the background and so a saturation increase of 50 percent was implemented. Additionally, the Hue Saturation Value (HSV) colour model was also used to try and segment skin hue due to its use in some of the literature as seen in Figure 4 but it proved to not be as adept at removing background noise as the YCbCr colour model. The final range of values determined for the YCbCr colour model range was Y (114,255), Cr (74,190) and Cb (147,204).

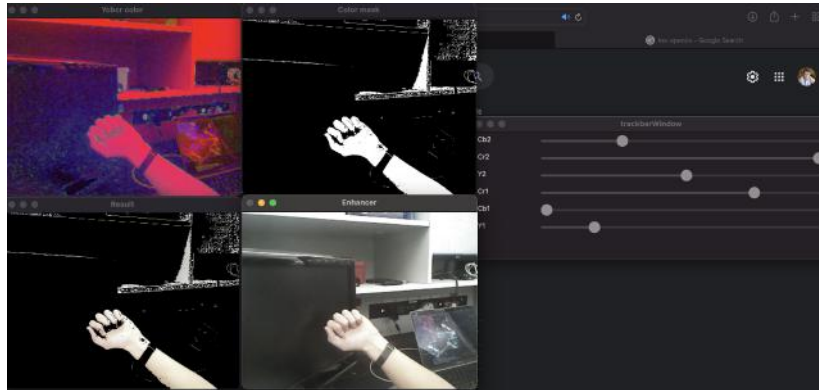


Figure 4.
Test program for varying HSV segmentation range.

The original design of the system involved a camera mounted above a table that could capture images of the user's hands and insert a virtual object into the scene. The user could then manipulate the object, and the coordinates of both the hand and virtual object would be simplified to just the width and height value of the image where they were located. A prototype was developed and the skin hue segmentation performed to segment just the user's hand and arm with a binary mask. Due to the black background of the table the segmentation was excellent and nearly noise-free barring a similar-coloured laptop case. The output of this prototype hand tracking is visible in Figure 5.

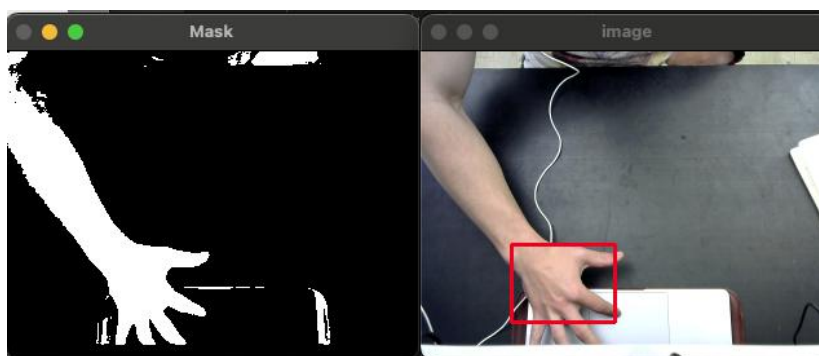


Figure 5.
Top-down view and skin hue segmentation.

However, it was later decided that in order to create a more realistic augmented reality scene

the camera should instead be mounted in front of the user so that another dimension was added to the scene and the user could "reach into" the frame and manipulate the virtual object as if it was in front of them rather than interacting through a removed third-person top-down view. This introduced more noise into the segmented image as some objects had similar hues to human skin and meant that the hand detection subsystem had to be overhauled.

As part of the hand detection preprocessing, the input image was first resized to 80 x 60 pixels as this allowed for faster processing due to the reduced dimensions. Skin hue segmentation was then performed and resulted in a noisy image shown in Figure 6 that contains the user's hand, arm and noise from objects in the image that have a similar hue to human skin.



Figure 6.
YCbCr segmented image and binary segmented image.

Taking inspiration from the classical approaches to hand segmentation, the design of the hand detection algorithm centred around determining which edge of the image the user's hand entered from and then finding the largest concentration of skin-coloured pixels closest to the opposite edge from that entry edge. This largest concentration of pixels represented the region in which the hand was most likely to be in because the hand is at the end of the arm and only skin-coloured pixels remain in the image. Experimentally, this proved to accurately detect the presence of a hand in an image, even when the hand was moved to different regions and the entry points of the arm varied widely.

The sum of the binary pixels present in each region were then calculated and the region's distance from the opposite edge of the hand's entry point calculated. The region closest to the opposite edge that contained over a certain threshold of binary pixels was then chosen as the region to most likely contain the hand. This threshold was experimentally determined to be 12000.

The detection of the user's hand needs to function when the hand is both far and near from the camera. Thus the bounding box to be drawn around the user's hand must be sized for the hand's closest view to the camera - within reason so as to not just estimate the whole image frame. Thus the size of the bounding box region was experimentally determined to be 20 x 15 pixels, which when moved over the 80x60 image by 5 pixels across first and then down results in a collection of candidate regions for hand presence. This was later increased to 1 pixel of movement in order to fit the region closer to the actual hand, providing 61 x 46 or 2806 candidate regions for hand detection. This is a high-enough resolution to both extract

the user's hand correctly and find its location to two decimal points of normalised location.

Furthermore, after system integration, the largest region of skin-coloured skin pixels closest to the opposite entry edge proved to be extracting the edge of the hand and not the whole hand as desired. A modification was made to the algorithm to find the largest region of skin-coloured pixels within a 8 pixel threshold of the edge-most region. This proved to extract the detected hand with increased accuracy and with little additional computational cost.

The region extracted by the algorithm - visible by the overlaid rectangle in Figure 7, is widened by 10 pixels on each side of the detected hand in order to extract the whole hand in cases where the detected region does not fully cover the extremities of the hand in the image. This extracted hand region is converted to grayscale and the minimum depth value from inside the hand region is extracted from the Kinect's depth imagery. These two variables are returned from the hand detection algorithm and provided as input to the gesture recognition classifiers.

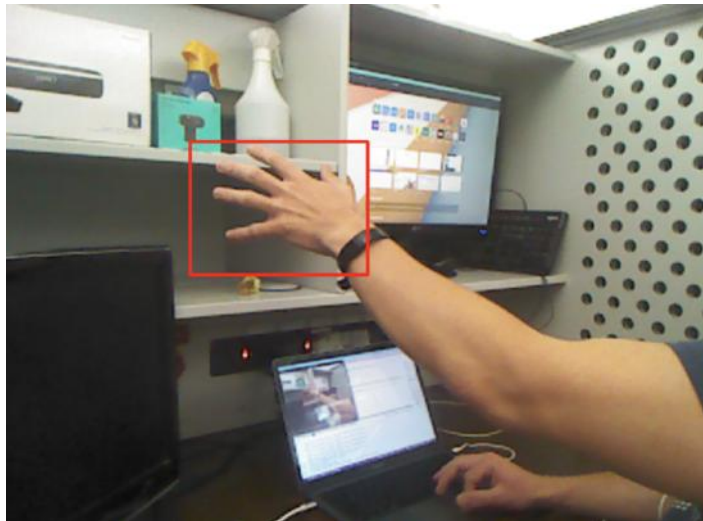


Figure 7.
The detected hand region shown by a red rectangle.

3.4 Neural network design

The rise of machine learning has led to the widespread adoption of neural networks for classifying and predicting non-linear behaviour and patterns. The specific use of CNNs in image classification has met with particular success and is the approach taken to develop the gesture recognition subsystem. The theory, design and implementation of the CNN is outlined in this section.

CNNs are traditionally comprised of a series of convolutional layers and pooling layers, the output of which is then connected to a fully-connected network consisting of hidden and output neurons. [27] The use of backpropagation with sets of training data and their desired output neuron values can be used to iteratively modify the internal parameters of the CNN to classify complex input images. The structure of a CNN is presented in Figure 8.

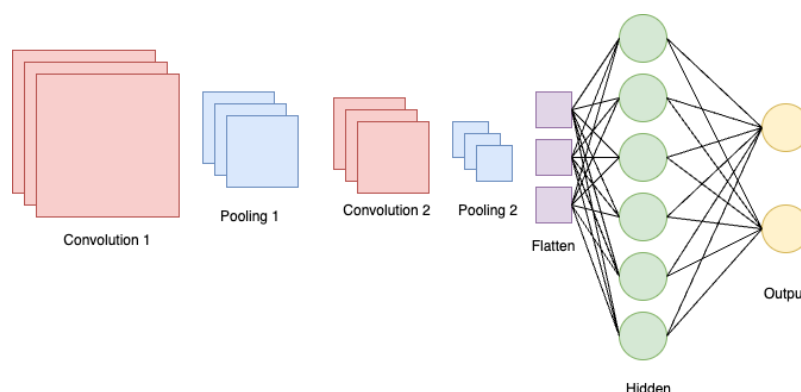


Figure 8.
Architecture of a CNN.

The convolutional layer and the convolution of a set of filters over an input image is the core mechanism by which a CNN functions. The filter or "kernel" is a matrix of values that when multiplied with the pixel values of a specific region of an input image, produces an output that effectively masks or modifies the input region. The operation of a convolutional filter as it slides over an input image can be seen in Figure 9. In traditional image processing, kernels can be used to blur, sharpen or detect the edges in an input image by applying the kernel to the successive regions of the image. CNNs work by letting the filter values be modified by the backpropagation algorithm during training to "learn" what features to produce as output of the convolution operation. These can be groups of certain coloured pixels, shapes or lines. When the convolution operation is applied to an input with multiple channels - like an RGB image, the result of the convolution for each channel is summed and set as the final output of the operation.

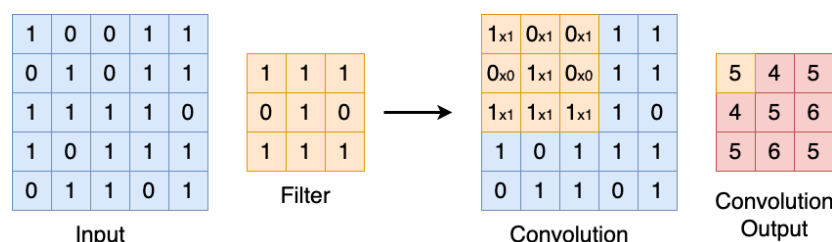


Figure 9.
The convolution operation of a filter over an input image.

Placing two convolutional layers after each other essentially allows a CNN to extract higher and higher-dimensional features of the image until the network is effectively extracting close-to-entire concepts like "fist" or "closed hand." An additional operation is performed by a pooling layer - inserted after a convolutional layer, this layer extracts either the average or maximum pixel value from a certain region and uses it to represent that portion of the region

in a smaller and less complex form. It effectively shrinks the size of the convolution output while still representing it accurately - this allows for increased computational performance.

Average pooling takes the average of a region of values while Max pooling takes the maximum value from a region of values. Max pooling is the preferred choice in most implementations due to the fact that by taking only the maximum value in a region it disregards the smaller noise values present there and improves performance on noisy inputs or noisy detected features. The operation of the two different kinds of pooling is shown in Figure 10. A parameter to also consider when conducting pooling and convolution is the stride size - by how many columns the filter or pooling region should slide over the input data. A larger stride size will mean a smaller output region but also has less ability to detect features that overlap the different sliding regions.

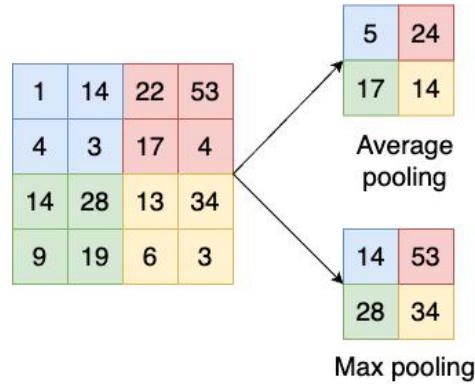


Figure 10.
The pooling operations on a region of input data.

By flattening the output of the final pooling layer and connecting it to a fully-connected layer of neurons, the CNN is now able to learn how to identify non-linear combinations of extracted shapes, groups of colours, edges and features. The weights between neurons and biases of each neuron in the fully-connected layer allow the network to learn - by itself, which specific detected features contribute more strongly to a certain classification and desired output of the network - according to provided training data.

The output of each neuron is the weighted sum of all its connected input neurons and the neuron's bias. When the weights connected to a neuron are specified by W_t and the bias by b , the output z of the neuron is given by

$$z = W_t \times x + b \quad (1)$$

and this output is "activated" by passing it through a nonlinear function such as a Sigmoid or Relu function in order to normalise the output to a certain data range and to introduce higher complexity into the network to allow it to learn non-linear patterns and classifications. The Relu function f applied to a neuron output z yields the output

$$f(z) = \max(0, z) \quad (2)$$

and thus returns the input if it is above 0 or else returns 0. The Sigmoid function returns the output

$$f(z) = \frac{1}{1 + e^{-z}} \quad (3)$$

and thus saturates the output for large values at 1 and for small values at 0. The Relu function is usually the activation function of choice in CNNs due to it thresholding values to 0 more often and thus causing the network to converge faster during training. The neurons in both the hidden and output layer are activated as well as the output of each convolution value in the convolutional layer. All of these processes take place in the forward propagation of the network as a new input is provided to the network and based on the values of the internal parameters, leads to a specific output.

The backpropagation algorithm is the algorithm that allows the network to "learn" and encapsulate decision-making behaviour in its modifiable parameters. The backpropagation algorithm works by propagating the error at the output of the network backwards so that the weights and biases of the fully-connected neurons and filter values in the convolutional layer can be updated to reduce that error at the output. Repeating this process with enough input data and desired output yields a network that can effectively classify input images into various learned categories.

Industry standard implementations of neural networks like Keras [28] envision the neural network as a connected series of functions - with each function accepting a multi-dimensional array as input and providing another as an output. In this way, chaining together the outputs of previous functions or "layers" as inputs for the following functions, a neural network can be constructed that propagates an input through a number of different operations using parameters learned during training to output a useful result. These functions or layers can be represented as their own classes and their internal logic abstracted away from using them in a network. This is the approach taken to designing the CNN used in the system and the overall class diagram for the system is presented in Figure 11.

The Network class is the data structure that represents the top-level interface of the CNN. It contains an array of layers that are the functions of the neural network and when modified can change the structure of the network as desired. The layers array is initialised with a number of the other layer classes. The forward propagation operation accepts an input array and then runs each of the layer functions consecutively with the output of the previous layer as its input in order to "propagate" an input through the network to the final output. The algorithmic description of the forward propagation operation is presented in Algorithm 2.

When the forward propagation operation is run for the first time the weights in the Hidden and Output classes are initialised using the He method described below and then the status of the network set to initialised. The prediction operation accepts an input, forward propagates it through the network and returns the output of the network. The backpropagation operation accepts a desired output for the network for a given input and then successively works backwards through the layers array, running the individual backpropagation operations for each layer and passing the back-propagated error to each previous layer and modifying the weights, biases and filter values where appropriate.

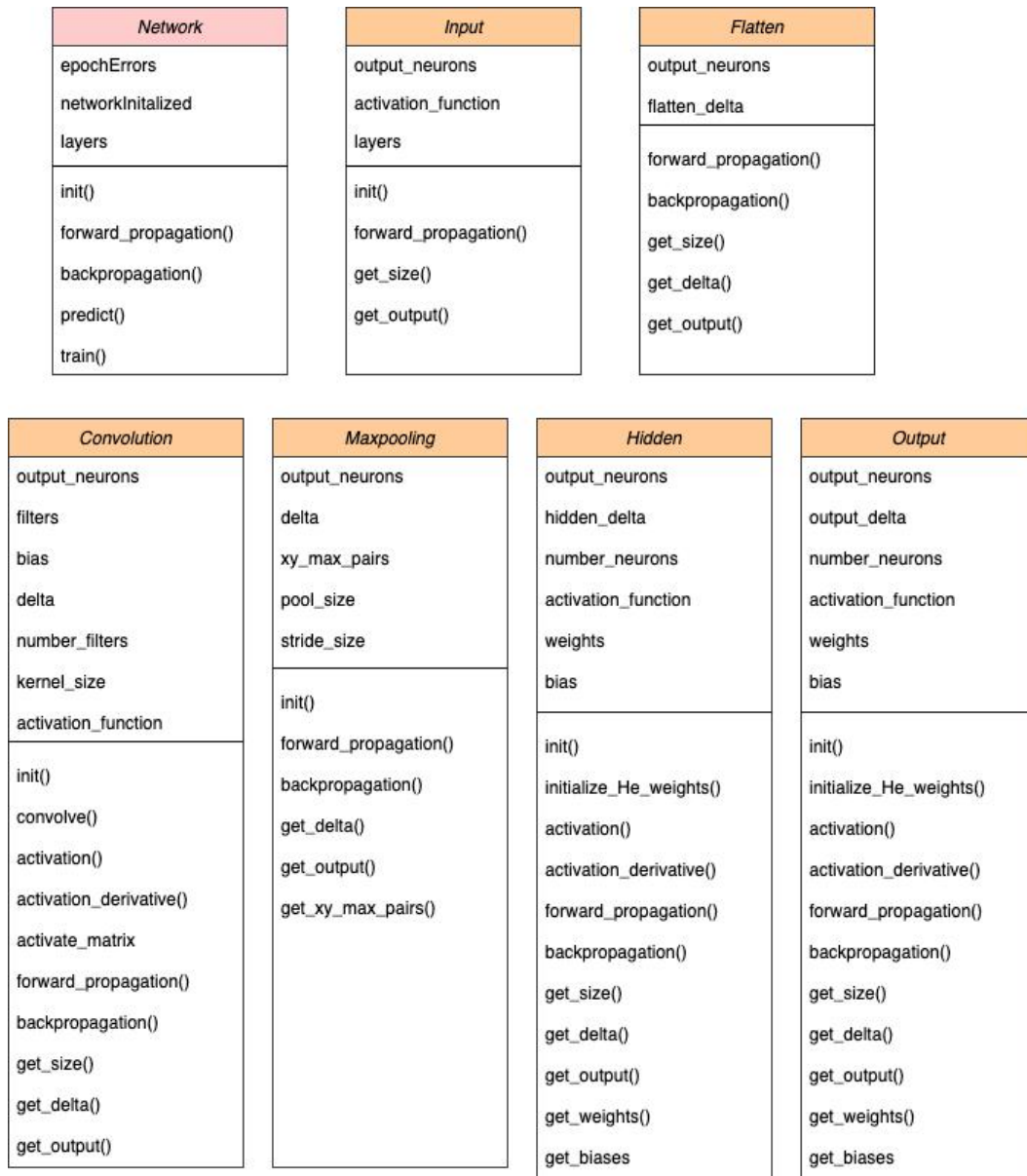


Figure 11.
Class diagram of the classes and operations used in the neural network.

The network's training operation accepts an array of training data input and array of training data output as well as hyperparameters such as learning rate and number of epochs to train over and then iteratively applies forward and backpropagation for each value in the training data in order to modify the parameters of the network to yield the desired classification behaviour. As this operation runs, the accumulative error that the network makes in classification is summed for every epoch and displayed to the user in order to show whether or not the network is improving at classification and at what rate.

Algorithm 2 Forward propagation of network

```

1: for layer in layers do
2:   if layer = 1 then
3:     forward propagate layer using new input of data
4:   else
5:     if network initialised then
6:       forward propagate layer using previous layer output
7:     else
8:       if layer hidden or output layer then
9:         initialise weights using He method
10:      end if
11:    end if
12:  end if
13: end for
14: set network_initialised = true

```

The Input layer is a simple class that merely accepts the input array and returns it - its existence is necessary in order to enable the Network class to be used to create either a convolutional or normal fully-connected neural network - this small Input class prevents code duplication and follows a more abstracted object-oriented approach.

The Convolution layer is used to create and extract features from the input image by sliding a filter across the input image pixels and is the centrepiece of the CNN and that allows it to identify and classify complex features in input images. The layer works by accepting input in its forward propagation operation and then for each filter and each channel in the input image, convolving the filter values with the pixels of the input image channel.

These convolved values are then activated using the Relu activation function and outputted in a multi-dimensional array. For the layer's backpropagation operation, the next layer's delta is used to calculate a filter gradient for each filter by convolving the input to the layer with the delta from the next layer and then modifying each filter by the filter gradient calculated for it. Additionally, the error or delta from the convolutional layer is calculated by performing a full convolution between the delta from the previous layer and versions of the filters in the layer that have been rotated 180 degrees. This delta is then returned and can be used by layers in front of it in the network to perform their own backpropagation.

When the system was integrated it was discovered that the naive convolution operation described above was one of the largest computational draws on the entire system and contributed a majority of the time taken to predict a gesture - preventing real-time operation of the system. Optimising this method became essential and thus the im2col method [29] was implemented to replace the series of nested for loops of the convolution operation with a matrix multiplication. This consisted of flattening the input array into a series of columns, matrix multiplying these with the flattened and repeated filter and then reshaping the output back into the required multi-dimensional shape. While the previous convolution operation took 27.77ms to convolve a 35x40 input array and a 9x9 kernel, the optimised method took only 3.01ms - an improvement of a factor of 9.

The Maxpooling layer is used to reduce the output size of an extracted convolution feature and reduce the amount of computations needed to process it effectively. The layer takes in an input array and iterates over it, returning the maximum value in each region of the input array - the size of the region being determined by pool size and stride size. Additionally, the location of the maximum value in each region is stored in a variable so that the value can be easily found and used by other layers during backpropagation. The Maxpooling layer itself has no parameters that are updated during training and thus the backpropagation method just returns the delta of the layer following it.

Similar to the convolution operation, the maxpooling operation required many nested for loops and was thus a computational bottleneck of the real-time system. Optimising this was achieved in a similar manner to the convolution operation, whereby a strided window was produced from the input array [30] and the maximum value found for each sub-array contained within the strided view. The original runtime for the maxpooling of a 27x32 input array with a pool size of 3x3 was 18.64ms and the optimised runtime was 1.53ms - a factor of 12 improvement.

The Flatten layer performs a simple function during forward propagation - it takes a multi-dimensional array and transforms it into a one-dimensional array. This reduces the complexity of the array and allows it to be connected to subsequent fully-connected Hidden layers that use weights and biases to learn what parts of an input array are important to a classification task.

Because the Flatten layer provides the input for a fully-connected hidden layer, it has to pass the error backwards from that Hidden layer during backpropagation. Thus in the backpropagation operation, the weights and biases for the next layer are used to calculate the error for the flattened layer by multiplying the errors with the next layer's delta and return it for use by previous layers.

The core element of the fully-connected neural network is the hidden layer and thus the Hidden class encapsulates this functionality and allows weights and biases between the layer and those following it to be modified in order to encode classification logic into the network. The He weights initialisation operation is used to randomly set the value of the weights and biases of a hidden layer to values that fall within a region specified by the standard deviation of the previous layer's size. This is done in order to prevent the weight values from being too large but also by covering the entire range of potential weight values with a normal distribution - giving extra probability that the weight value being generated will be closer to 0. This is to reduce training time and not saturate the activated neurons when using the Relu activation function.

The forward propagation operation in the hidden layer accepts the previous layer's output as input and then for each neuron in the layer's output, calculates the dot product between the previous layer's output and all the weights connected to that neuron in the layer. The bias is added to the dot product and then the whole value "activated" using the activation method which applies the Relu or Sigmoid activation function to the value depending on the parameters of the layer. This is done for each neuron and thus the input is propagated forward through the network using the weights and biases as parameters.

Those parameters are then updated during the backpropagation method where the error

from the previous layer is accepted as input and the error for each neuron is calculated by multiplying the weight connecting the neuron to the neuron in the next layer with the value of the error in that next layer neuron. The delta for each neuron is then calculated by multiplying the error associated with it with the derivative of the activation function of that neuron's value. The weight is then updated by multiplying its value with the delta value and previous layer's output for each neuron connected to it. The bias values are updated by just multiplying them with their calculated deltas.

In this way, the error is propagated backwards through the network and the network "learns" from desired output given to it as it adjusts its internal parameters towards outputting desirable results. The Output class operates identically to the Hidden class because it is also a fully-connected collection of neurons; however the errors calculated for each neuron are merely the desired output for each neuron subtracted from the current output of the neuron. Additionally, the neurons in the layer are activated using the softmax activation function in order to provide a probabilistic output of whether or not a certain pattern is present in the network.

By combining all of the layers described above into a successive computation that is repeated using various input data and desired output values, a CNN can be created that accepts image data as input and provides probabilistic classifications as output. The combination of discrete components that encapsulate the computations for Input, Convolution, Maxpooling, Flatten, Hidden and Output layers in an overall Network class allow for the easy modification of neural networks and complex decision-making to be implemented in software.

3.5 Neural network training and testing

In order to capture training data for the various classifiers needed for the gesture recognition subsystem, a small program was developed to take 5 pictures per second using the Kinect sensor and store them in a folder until the user closes the program. This follows a previous program that required the user to press the spacebar to take each picture and although it worked - took a very long time to generate a large dataset and led to edge cases that the network could not correctly classify. By generating a larger dataset with the hand gestures being moved throughout the entire frame of the image the accuracy of the classifiers was increased.

What became apparent during initial testing of the first-principles backpropagation code, is that it was incredibly slow. Training a network with 1 convolutional layer with 3 filters, a Maxpooling layer, a hidden layer with 100 neurons and 10 output neurons took on average 0.91 seconds per epoch for a training dataset of 10 28x28 images of hand-written digits.

Scaling this up to a dataset of 1000 images, the expected time to train a network over 5 epochs is roughly 4550 seconds or over an hour and if the image size is increased to a 60x60 image for more resolution as is expected to be necessary, the expected time to train the already small network with 1000 images increases to 5.28 seconds per epoch as seen in Figure 12 or over 7 hours for 5 epochs. Figure 12 also coincidentally shows the ability of the first-principles neural network to learn complex input data and converge towards a model that can accurately classify input data.

```

Epoch: 0, Learning rate: 0.01, Error: 43.505664429719, Runtime: 5.44 seconds
Epoch: 1, Learning rate: 0.01, Error: 35.90939001147681, Runtime: 5.26 seconds
Epoch: 2, Learning rate: 0.01, Error: 30.71071901930605, Runtime: 5.25 seconds
Epoch: 3, Learning rate: 0.01, Error: 26.40187856086965, Runtime: 5.25 seconds
Epoch: 4, Learning rate: 0.01, Error: 22.7233303540696, Runtime: 5.26 seconds
Epoch: 5, Learning rate: 0.01, Error: 19.59382213497107, Runtime: 5.26 seconds
Epoch: 6, Learning rate: 0.01, Error: 17.033672088044533, Runtime: 5.33 seconds
Epoch: 7, Learning rate: 0.01, Error: 14.977637472115275, Runtime: 5.27 seconds
Epoch: 8, Learning rate: 0.01, Error: 13.306794349399123, Runtime: 5.33 seconds
Epoch: 9, Learning rate: 0.01, Error: 11.929754574796652, Runtime: 5.24 seconds
Epoch: 10, Learning rate: 0.01, Error: 10.786391514780357, Runtime: 5.24 seconds

```

Figure 12.

Output of the first principles backpropagation algorithm when using a 60x60 image.

This is an extremely long period of time considering that tens of training iterations would have to be completed for each type of gesture classifier in order to optimise it. Thus the use of the TensorFlow library [31] for training the classifiers was undertaken because of the library's optimised nature and ability to run backpropagation much more efficiently than the first principles implementation. Training the same network with TensorFlow and the same 60x60 input images took 170ms per epoch on average - a 31-times improvement over the first principles implementation.

In order to properly test and validate the performance of the neural network models developed for gesture classification, proper splitting of the datasets used to train each model had to be performed. Thus, any data that was used to train a model was first randomised in order to not over-train the model to a specific set of features of one of the input classes and the last 10% of this randomised dataset reserved only for testing the accuracy of the model at the end of training. Furthermore, the last 10% of the training dataset was reserved for testing the validation accuracy of the model during training - this is in-line with best practices in industry that use validation accuracy as a way of fine-tuning model hyperparameters such as network layer size of number of hidden neurons. Once the model has been trained and tested, it is used to classify live frames from the Microsoft Kinect camera and a subjective opinion formed if the model developed can actually classify gestures correctly enough of the time to make real-time gesture control a reality.

Turning now to the network architecture used to classify gestures, some approaches in the literature [12] took in an RGB image as input and through a large collection of convolutional operations and hidden layers, outputted coordinates for landmarks of the hand such as top of the fingers and edges of the palm. A naive implementation of this was implemented by attempting to output a coordinate for an input image of the hand where the pinky finger in that image was located. A point was selected in each training input image where the pinky finger was located. The delta of each pixel in the image from the pixel where the pinky finger was located was then calculated to produce a heatmap of the image with regards to pinky location. This heatmap is visible in Figure 13. This heatmap was then passed into a CNN which was attempted to be trained to output the coordinates of the pinky.

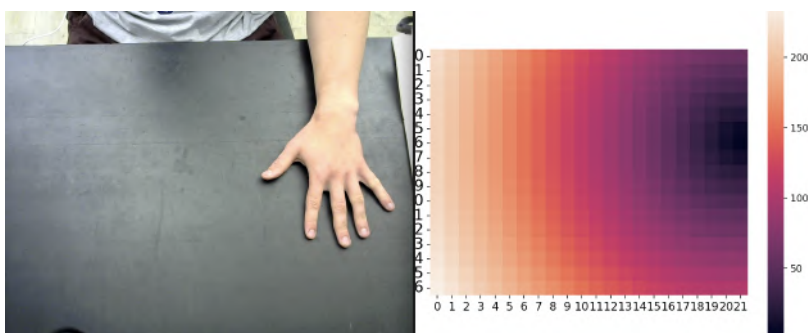


Figure 13.
Heatmap for naive pinky detector.

However the output of the network could never reliably achieve this no matter how many layers or parameters were introduced into the network. The accuracy rating never reached above 50% - no better than chance or noise. This is hypothesised to be due to CNNs being invariant to position and not being the ideal data structure to predict specific values like pixel locations - they are built more for classification and prediction. Complex intermediate transformations were used in the literature approaches studied in between the convolutional layers and the fully-connected part of the network and thus trying to achieve a small subset of the same results without those intermediate calculations proved to be unsuccessful and the implementation of them deemed too computationally burdensome for an already processing-strained system. Thus a simpler approach that merely classified input images into different categories like "open hand" or "fist" was pursued instead and is documented in Section 3.6.

In order to compare the different speed performance of various neural network architectures, a grid search was performed that measured the time taken for a single inference of a basic CNN for classifying whether a hand was open or closed - thus containing two output neurons. The image size was 80x60 pixels and the frames per second that would be achieved for the classifier calculated as well. The time given for a single inference was an average of the first 100 inferences of the network on a live input image.

The results are shown in Table 2 and show that the fewer convolutional layers present in the network, the faster it performed - 0.0471 seconds for a network with one convolutional layer and 0.0634 seconds for a network with two, both with kernel sizes of 3x3 and 500 hidden neurons.

Number Convolution Filters	Kernel Sizes	Number Hidden Neurons	Runtime (s)	FPS
16,32	(9x9),(3x3)	100	0.0767	13.0334
16,32	(3x3),(3x3)	100	0.0630	15.8617
16,32	(3x3),(3x3)	50	0.0575	17.3810
16,32	(3x3),(3x3)	500	0.0634	15.7619
16,32	(3x3),(3x3)	1000	0.0686	14.5598
16	(3x3)	100	0.0386	25.8411
8	(3x3)	100	0.0196	50.9814
16	(3x3)	500	0.0471	21.2229
32,64	(3x3),(3x3)	50	0.1455	6.8719
8,16	(3x3),(3x3)	50	0.0277	36.0956

Table 2.

Runtime duration for various hand classifier network configurations (Average of 100 classifications).

The networks with more hidden layers took slightly longer to complete all their calculations as expected due to more dot products needing to be computed between each hidden layer neuron and the flattened inputs before it and output neurons following it. This is shown by the 0.0471 seconds it takes the network with 500 hidden neurons to complete an inference as opposed to the same network but with just 100 neurons that took 0.0386 seconds. The difference in runtime is quite small however and is due to the optimised NumPy [25] dot product method being used to compute the calculations in the hidden layer.

The largest difference in runtime came from reducing the number of convolution filters in each convolutional layer. A convolutional layer of 16 filters took 0.0386 seconds to run as opposed to a 0.0196 runtime for 8 filters. Thus the fewer number of both convolutional layers and filters in those layers that can be used to achieve good accuracy in a network - the better it will perform. Additionally, a smaller number of hidden neurons and smaller kernel sizes in the convolutional layers will yield additional performance.

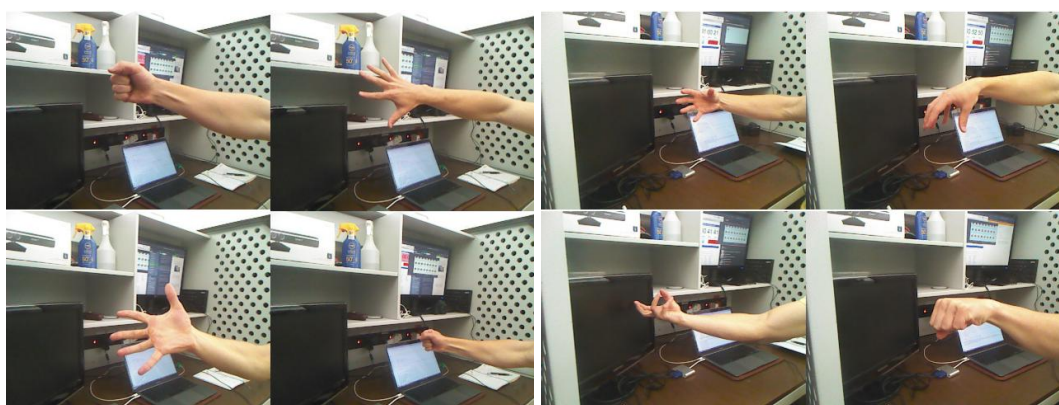
3.6 Hand gesture recognition

The development of the hand gesture recognition subsystem made extensive use of the first principles neural network data structure. Training was conducted for the gesture recognition networks using TensorFlow for the speed and efficiency reasons outlined in Section 3.5. In order to model the current gesture of the user's hand accurately enough to infer the necessary different virtual object commands, the following classifiers were developed.

An open hand or closed fist classifier was developed to determine whether the virtual object has been grasped by the user and should now be attempted to be moved to the user's hand

position. This was developed and trained on a dataset of 1600 captured images, a subset of which is visible in the first part of Figure 14. A classifier was developed that determined if the user's hand pointed downwards, to the side or upwards all relative to the camera - this would be used to control the virtual object's rotation about the z axis. It was trained on a dataset of 4200 images, a subset of which is visible in the second part of Figure 14. A classifier was developed that determined if the user's hand was pointing to the left, towards or right of the camera and was used to control the rotation of the virtual object about y axis. It was trained on a dataset of 900 captured images, a subset of which is visible in the first part of Figure 15.

Finally, a classifier was developed that could determine if the user's hand was pointing downwards, forward or upwards relative from the side-on view of the camera and was used to control the rotation of the virtual object around the x axis. It was trained on a dataset of 1800 captured images, a subset of which is visible in the second part of Figure 15. These four classifiers allow the system to recognise 11 different gestures and use these gestures to send commands to the virtual object without the computational overhead of building a much larger single classifier to recognise and differentiate between all the different gestures.



Subset of Open, Fist dataset

Subset of Down, Side, Up dataset

Figure 14.

Subsets of the datasets used to train the Open, Closed classifier and the Down, Side, Up classifier.

The performance of the different classifiers varied widely based on the architecture of the CNN used to construct them as well as the hyperparameters used during training. Thus a grid search was performed to establish the performance differences of the various architectures with regards to the training accuracy, training validation accuracy as well as testing accuracy for each classifier. This grid search is reminiscent of a more directed Monte Carlo search through the potential solution space - obviating the extreme ranges of the solution space where performance was already shown to be declining before this point and in the interests of training time and computational limitations.



Subset of Left, Towards, Right dataset

Subset of Down, Forwards, Up dataset

Figure 15.

Subsets of the datasets used to train the Left, Towards, Right classifier and the Down, Forwards, Up classifier.

The purpose of the grid search also serves as a way to document the ability of the CNN to solve the problem of gesture recognition in various different configurations which will be useful when attempting to integrate the subsystem into a real-time system that requires its subsystems to operate in a certain timeframe. In order to meet a real-time operation specification, the system may need to use a slightly less optimal classifier in terms of accuracy but one that achieves similar performance but in a much more computationally efficient manner. The grid searches were all conducted with a learning rate of 0.001 and the epoch result shown for each classifier the best training epoch of the nearby search space.

For the Open hand, closed fist classifier grid search shown in Table 3, the architecture with the most desirable performance is made up of two convolutional layers with 16 and 32 filters each, kernel sizes of 9x9 and 3x3 respectively as well as 1000 neurons in the hidden layer. The use of a larger kernel size is notable and is hypothesized to lead to increased performance due to the larger width an open hand occupies in an image compared to other gestures that are more finely-differentiated. The larger kernel size allows the network to more easily detect patterns that make up an open hand and thus the network performs better. This architecture achieves a 95.91% accuracy, 94.44% validation accuracy and 96.25% test accuracy when trained over 6 epochs. The accuracy and loss graphs for this classifier are shown in Figure 16.

Number Convolution Filters	Kernel Sizes	Number Hidden Neurons	Epochs	Accuracy	Validation Accuracy	Test Accuracy
16	(3x3)	1000	10	0.9699	0.9167	0.9437
16	(3x3)	1000	4	0.9761	0.9097	0.8562
16	(3x3)	500	4	0.9792	0.9306	0.9375
16	(3x3)	100	4	0.9684	0.9028	0.9312
16,32	(3x3),(3x3)	1000	4	0.9498	0.9722	0.9499
16,32	(3x3),(3x3)	500	4	0.9421	0.9306	0.9125
16,32	(3x3),(3x3)	500	6	0.9560	0.8819	0.9562
16,32	(3x3),(3x3)	100	4	0.9676	0.9236	0.9062
16,32	(3x3),(3x3)	100	6	0.9653	0.9444	0.9562
16,32	(3x3),(3x3)	50	4	0.9614	0.9375	0.9250
16,32	(3x3),(3x3)	50	6	0.9676	0.9514	0.9250
16,32	(9x9),(3x3)	1000	6	0.9591	0.9444	0.9625
16,32	(9x9),(3x3)	500	6	0.9506	0.9444	0.9562
16,32	(9x9),(3x3)	100	6	0.9630	0.9236	0.9562
16,32	(9x9),(3x3)	50	6	0.9653	0.9236	0.9437
16,32	(9x9),(3x3)	50	4	0.9537	0.9375	0.9125
16,32	(9x9),(3x3)	25	6	0.9591	0.9444	0.9562
16,32	(9x9),(3x3)	25	4	0.9414	0.9514	0.9499

Table 3.

Grid search of Open, Fist hand classifier hyperparameters using a learning rate of 0.001.

Number Convolution Filters	Kernel Sizes	Number Hidden Neurons	Epochs	Accuracy	Validation Accuracy	Test Accuracy
16,32	(9x9),(3x3)	1000	6	0.9583	0.9583	0.9562
16,32	(9x9),(3x3)	100	6	0.9468	0.9375	0.9562
16,32	(3x3),(3x3)	1000	6	0.9722	0.9444	0.9062
16,32	(3x3),(3x3)	100	6	0.9482	0.8819	0.9125

Table 4.

Grid search of Open, Fist hand classifier hyperparameters using a learning rate of 0.0001.

Modifying the learning rate from 0.001 to 0.0001 had a small impact on performance as evident in Table 4 where the various accuracies change by a percentage point based on the

learning rate. This is not enough gain to consider experimenting further with different learning rates.

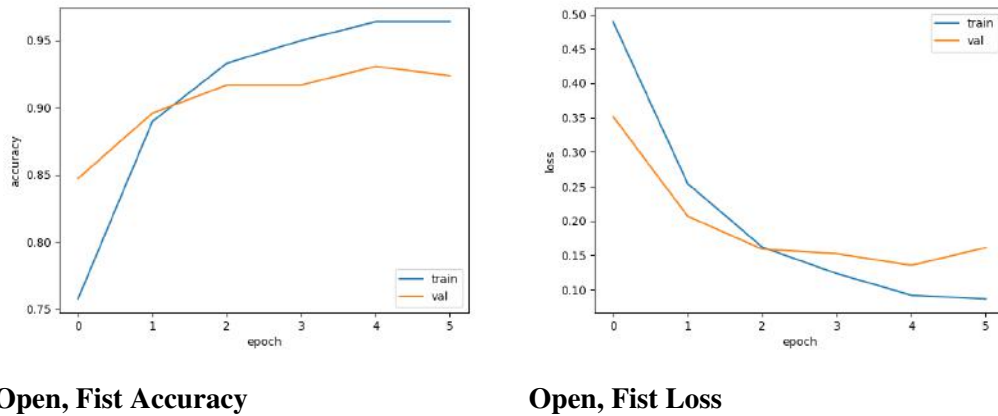


Figure 16.
Graphs showing the accuracy and loss of the Down, Forwards, Up Classifier during training.

The conclusion of the grid search in Table 5 is that in order to accurately classify if a hand is pointing to the left, right or towards the camera more than 500 hidden neurons is counter-productive. This is due to the networks using more than 500 neurons in the hidden layer all having a test and validation accuracy of below 93.33%. The number of neurons used in the hidden layer tends to be inversely correlated with testing and validation accuracy - the fewer neurons that are used in the hidden layer the better the network performs when classifying previously unseen data. The networks using kernel sizes of 9x9 tend to perform better as well. This is a clear example of how too large a network can lead to overfitting on the training data and not be able to generalise to the patterns in training data so the system actually has an intuitive understanding of the classes it is classifying.

Two of the best classifiers rely on one convolutional layer - showing that the simpler network is better at generalising the shape of the left, towards and right-pointing hand. The best performing classifier is selected from these two and is the single convolution layer with 16 filters, a kernel size of 9x9 and 100 hidden neurons trained over 6 epochs. This classifier has an accuracy of 95.88% but more importantly, a validation and test accuracy of 96.30% and 97.77% respectively. The accuracy and loss curves for the 6 training epochs are presented in Figure 17.

Number Convolution Filters	Kernel Sizes	Number Hidden Neurons	Epochs	Accuracy	Validation Accuracy	Test Accuracy
16,32	(3x3),(3x3)	1000	5	0.9575	0.9630	0.9333
16,32	(3x3),(3x3)	1000	7	0.9369	0.9753	0.9333
16,32	(3x3),(3x3)	800	6	0.9534	0.9259	0.9222
16,32	(3x3),(3x3)	500	6	0.9396	0.9877	0.9333
16,32	(3x3),(3x3)	300	6	0.9643	0.9259	0.9777
16,32	(3x3),(3x3)	200	6	0.9383	0.9630	0.9444
16,32	(3x3),(3x3)	100	6	0.9410	0.9012	0.9555
16,32	(3x3),(3x3)	50	6	0.9657	0.9383	0.9555
16	(3x3)	100	6	0.9588	0.9630	0.9666
16	(3x3)	200	6	0.9698	0.9383	0.9111
32	(3x3)	100	6	0.9630	0.9506	0.9111
16	(9x9)	100	6	0.9588	0.9630	0.9777
32	(9x9)	100	6	0.9671	0.9506	0.9111
16,32	(9x9),(3x3)	100	6	0.9479	0.9877	0.9222
16,32	(3x3),(3x3)	50	10	0.9643	0.9383	0.9555
16,32	(3x3),(3x3)	25	10	0.9588	0.9259	0.9333

Table 5.
Grid search of Left, Towards, Right hand classifier hyperparameters.

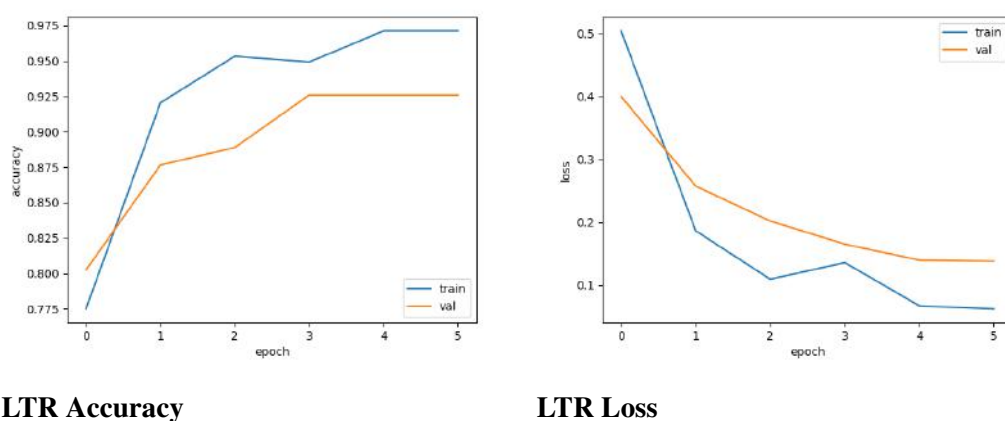


Figure 17.
Graphs showing the accuracy and loss of the Left, Towards, Right Classifier during training.

Number Convolution Filters	Kernel Sizes	Number Hidden Neurons	Epochs	Accuracy	Validation Accuracy	Test Accuracy
16,32	(3x3),(3x3)	500	5	0.9986	0.9938	1.0
16,32	(3x3),(3x3)	100	5	1.0	1.0	0.9944
16	(3x3)	100	5	1.0	1.0	0.9888
16	(3x3)	50	5	0.9952	0.9877	0.9944
8,16	(3x3),(3x3)	50	5	0.9986	1.0	1.0

Table 6.
Grid search of Down, Forwards, Up hand classifier hyperparameters.

The Down, Forwards and Up classifier for which the architecture grid search is shown in Table 6 was much easier to train. This is due to the simpler nature of the classifier - if the image contains more skin-coloured pixels in the bottom half of the image it is probably a downwards-facing hand, if there are more skin-coloured pixels in the top half of the image it is probably an upwards-facing hand and if there are more skin-coloured pixels in the middle of the image the hand is probably pointing to the side. It is also hypothesised that the dataset used to train the classifier contained fewer edge cases of confusing hand gestures with the fingers splayed out and the more straight hands provided to the classifier allowed training to be performed more easily. This highlights the importance of clean and representative data when training neural networks.

The classifier reaches over 99% accuracy, validation accuracy and test accuracy when trained over 5 epochs for all but one of the tested architectures. This is a positive sign for shrinking the network so that it performs faster while still retaining good performance. Thus the ideal network architecture here is a single convolutional layer with 16 filters, a kernel size of 3x3, 100 hidden neurons and that is trained over 5 epochs. It has an accuracy of 100%, and validation and test accuracy of 100% and 98.88% respectively. The accuracy and loss curves for the 5 training epochs are presented in Figure 18.

Finally, the Down, Side, Up classifier grid search shown in Table 7 shows that the best-performing network architecture is the network comprised of two convolutional layers with 16 and 32 filters each, kernel sizes of 9x9 and 1000 neurons in the hidden layer. This network achieves a 97.80% accuracy, 98.68% validation accuracy and 98.33% test accuracy when trained over 5 epochs. The accuracy and loss curves for the 5 training epochs are presented in Figure 19 and show a slight decrease in validation accuracy at the end but offset by a gain in training accuracy. This is a tradeoff made with the intention of relying on real-time input data to the classifier being similar to the many varied training inputs.

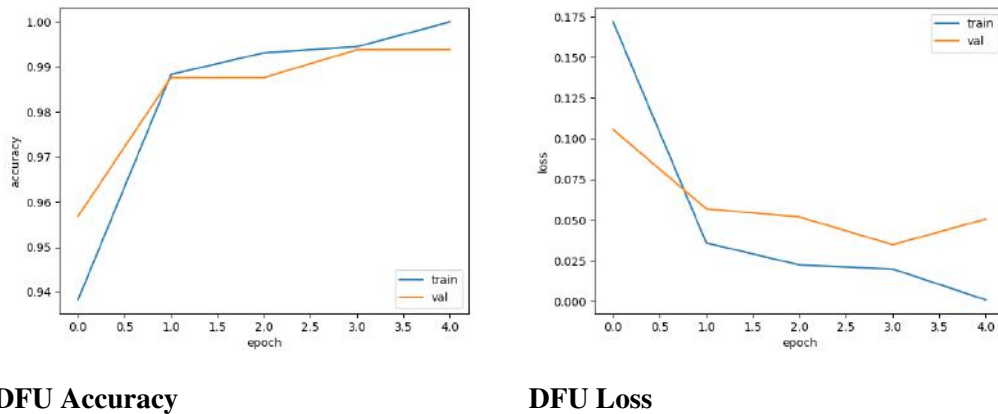


Figure 18.
Graphs showing the accuracy and loss of the Down, Forwards, Up Classifier during training.

Number Convolution Filters	Kernel Sizes	Number Hidden Neurons	Epochs	Accuracy	Validation Accuracy	Test Accuracy
16,32	(9x9),(9x9)	1000	5	0.9780	0.9868	0.9833
16,32	(9x9),(9x9)	500	5	0.9815	0.9735	0.9714
16,32	(9x9),(9x9)	100	5	0.9874	0.9444	0.9523
16,32	(3x3),(3x3)	500	5	0.9856	0.9947	0.9761
16,32	(3x3),(3x3)	100	5	0.9844	0.9603	0.9595
16,32,64	(3x3),(3x3)	100	5	0.9777	0.9392	0.9357

Table 7.
Grid search of Down, Side, Up hand classifier hyperparameters.

What is more important than incremental gains in validation or test accuracy is the performance of the system on real input from the Microsoft Kinect sensor. If a gesture classifier incorrectly classifies a gesture once or twice it is negligible to the overall performance of the system if the gesture is then correctly classified in the next few frames and the user seemingly able to manipulate the virtual object as desired. More detailed insight into this real-world performance of the classifiers is provided in Section 4.

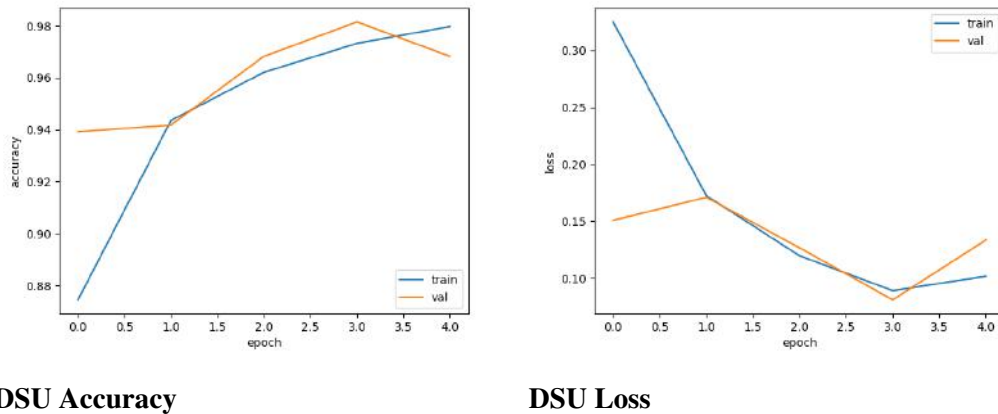


Figure 19.
Graphs showing the accuracy and loss of the Down, Forwards, Up Classifier during training.

3.7 Surface detection

In order to accurately identify the surface of the table in the video frame and allow the virtual object to rest on the table as well as prevent the object from passing through it, some form of plane estimation must be implemented. This is so that the entire surface of the table can be found at the start of the program when the frame is empty and the surface is clearly visible and all the pixels that form part of the table can be marked as such and their depth values evaluated in the depth image from the Microsoft Kinect. In this way, the virtual object's position and depth can later be compared to the table surface's depth and position values and interactions between them modelled. Plane estimation is a well-studied problem and the approach taken to implement it is by taking inspiration from Yang [8] and using a modified implementation of the RANSAC algorithm for finding planar surface using normal vectors.

The normal RANSAC implementation picks a random collection of pixels in an image and for each pixel calculates the normal vector and thus plane that pixel lies on and is a part of. The surrounding pixels are then successively evaluated and if their normal vectors match the original pixel they are grouped together as part of the same plane. The algorithm iteratively "grows" the planar surfaces these points are a part of until all pixels in the image have been accounted for and a collection of planar surfaces and their constituent pixels are stored in memory. This approach accounts for every planar surface present in the image and identifies a plane for every pixel in the image. This is unnecessary for the detection of just the table surface and thus the algorithm is modified to only begin with one single pixel of the table surface and then to grow the associated planar region until no more pixels can be found that form part of the plane. In this way the table surface is accurately found with much less computation than the full RANSAC implementation.

The single starting pixel is selected by a form of calibration at the startup of the overall system.

A live video feed of the scene is displayed to the user with a rectangle overlaid near the bottom of the screen where the table surface is expected to be. This calibration screen is visible in Figure 20. The user is then prompted to move the camera until the rectangle covers some part of the surface of the table and then press the spacebar key to begin the calibration. The pixel found in the middle of this rectangle at width and height 360 and 410 pixels respectively is chosen as the starting pixel. Its depth value is extracted from the depth input image and its normal computed and stored as the chosen normal.



Figure 20.
The surface detection calibration screen shown to the user at the start of the program.

The normal vectors for every pixel in the image are then computed and stored. The normal vector calculation for a pixel relies on the pixels above and below as well as to the right and left of the original pixel. This is because the normal vector for a point is perpendicular to the orthogonal x and y tangent vectors to the plane the point rests on. [32] Mathematically, consider a point $P = (p_x, p_y, p_z)$ in three dimensions, where $p_z = z(x, y)$ a function that computes the depth value from the x and y points. Figure 21 shows this visually. The orthogonal tangent vectors of this point with respect to the x and y axes are

$$\frac{\partial p}{\partial x} = \left(\frac{\partial p_x}{\partial x}, \frac{\partial p_y}{\partial x}, \frac{\partial p_z}{\partial x} \right) = \left(1, 0, \frac{\partial z}{\partial x} \right) \quad (4)$$

and

$$\frac{\partial p}{\partial y} = \left(\frac{\partial p_x}{\partial y}, \frac{\partial p_y}{\partial y}, \frac{\partial p_z}{\partial y} \right) = \left(0, 1, \frac{\partial z}{\partial y} \right) \quad (5)$$

respectively.

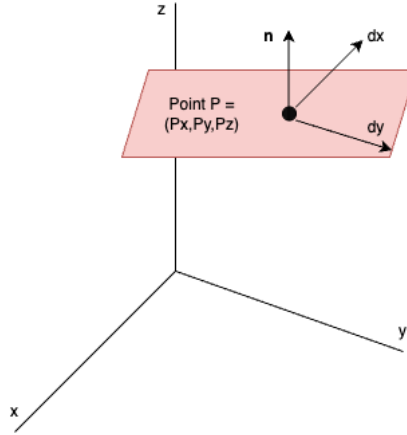


Figure 21.
A point on a plane with its normal and gradients expanded.

The gradient of p_z with regards to the x and y axes can be found by computing the gradient of the points on either side of the original point $p_{z+1} = z(x+1, y)$ and $p_{z-1} = z(x-1, y)$ with regards to the x axis and $p_{z+1} = z(x, y+1)$ and $p_{z-1} = z(x, y-1)$ with regards to the y axis. Mathematically, this can be seen by

$$\frac{\partial z}{\partial x} = \frac{z(x+1, y) - z(x-1, y)}{(x+1) - (x-1)} \quad (6)$$

$$= \frac{z(x+1, y) - z(x-1, y)}{2} \quad (7)$$

and

$$\frac{\partial z}{\partial y} = \frac{z(x, y+1) - z(x, y-1)}{(y+1) - (y-1)} \quad (8)$$

$$= \frac{z(x, y+1) - z(x, y-1)}{2} \quad (9)$$

The normal vector for the point P is calculated as the cross-product of the two orthogonal tangent vectors to the plane the point lies on and is calculated as

$$\mathbf{n} = \frac{\partial p}{\partial x} \times \frac{\partial p}{\partial y} \quad (10)$$

$$= \left(1, 0, \frac{\partial z}{\partial x}\right) \times \left(0, 1, \frac{\partial z}{\partial y}\right) \quad (11)$$

$$= \left(0 \times \frac{\partial z}{\partial y} - \frac{\partial z}{\partial x} \times 1, \frac{\partial z}{\partial x} \times 0 - 1 \times \frac{\partial z}{\partial y}, 1 \times 1 - 0 \times 0\right) \quad (12)$$

$$= \left(-\frac{\partial z}{\partial x}, -\frac{\partial z}{\partial y}, 1\right). \quad (13)$$

In order to compare normal vectors to one another it is necessary to convert the normal vector to unit normal vector. This is achieved by dividing the normal vector by its magnitude. The magnitude can be calculated as

$$mag = \sqrt{\left(-\frac{\partial z}{\partial x}\right)^2 + \left(-\frac{\partial z}{\partial y}\right)^2 + (1)^2} \quad (14)$$

and the final unit normal

$$\mathbf{n} = \left(\frac{-\frac{\partial z}{\partial x}}{mag}, \frac{-\frac{\partial z}{\partial y}}{mag}, \frac{1}{mag} \right). \quad (15)$$

Using the above calculations, the system calculates the normal vector for every pixel in the image. Using the pixel in the middle of the calibration region rectangle as the starting pixel, the modified RANSAC algorithm can then be used to find the plane of the table. Arrays of rows and columns representing the surface pixel rows and columns are created. Then the surface of the table is determined by checking the pixels to the left, right, top and bottom of the current pixel and if their normal vectors are similar to the starting pixel's normal vector. If they are similar, the pixel is added to the surface arrays.

The algorithm then increments a counter and considers the next row and column pair from the arrays and determines if they are a part of the surface by comparing their normal vectors. The process is repeated until no more values are added to the arrays and thus all the surface normals that are similar to the starting pixel's normal vector and are adjacent to the other surface pixels are found. The algorithmic description of the surface calibration process is provided in Algorithm 3.

In this way, the planar surface of the table is found and the result of this algorithm can be seen in Figure 22 where the extracted surface is shown in white on the left and the calculated normals for the entire image on the right - with the normal vectors of the table surface being displayed in purple.

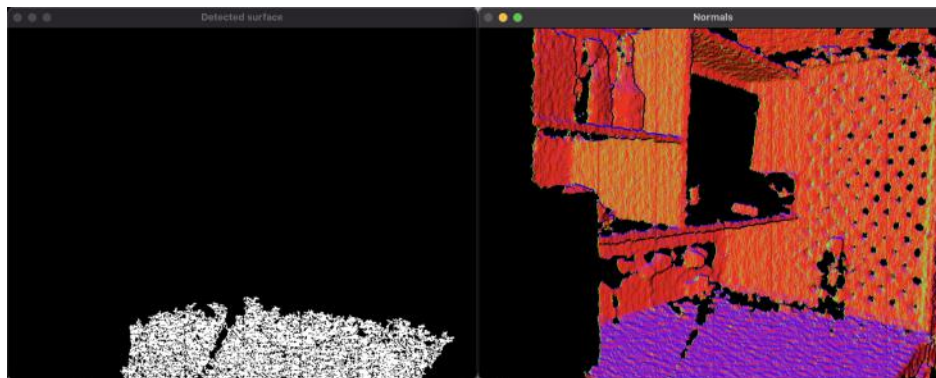


Figure 22.
The result of the table surface detection and extraction algorithm.

Algorithm 3 Surface calibration

```

1: read in new Kinect RGB and depth data
2: for each pixel in depth image do
3:   calculate normal vector
4:   calculate normal vector magnitude
5:   divide normal vector by magnitude to get unit normal vector
6: end for
7: select pixel in centre of calibration rectangle as starting pixel
8: set surface_pixels to empty array
9: set chosen_normal to unit normal at centre of calibration rectangle
10: pixel_counter = 0
11: while pixel_counter < length(surface_pixels) do
12:   if pixel to right has normal close to chosen_normal then
13:     if right pixel not in surface_pixels already then
14:       add right pixel to surface_pixels
15:     end if
16:   end if
17:   if pixel to left has normal close to chosen_normal then
18:     if left pixel not in surface_pixels already then
19:       add left pixel to surface_pixels
20:     end if
21:   end if
22:   if pixel above has normal close to chosen_normal then
23:     if above pixel not in surface_pixels already then
24:       add above pixel to surface_pixels
25:     end if
26:   end if
27:   if pixel below has normal close to chosen_normal then
28:     if below pixel not in surface_pixels already then
29:       add below pixel to surface_pixels
30:     end if
31:   end if
32:   pixel_counter = pixel_counter + 1
33: end while
34: return surface_pixels

```

While testing the surface detection algorithm a weakness of the Microsoft Kinect sensor was discovered. The Kinect computes depth values by projecting a speckle pattern using infrared light onto the scene in front of it. The deformation of the speckle pattern is then identified using an infrared camera and the depth values of each pixel in the environment computed. However, certain surfaces are better absorbers of infrared radiation than others - in particular black matte surfaces absorb infrared more readily than light shiny surfaces. The rubber surface of the tabletop the surface detection system was being tested on in the Project Lab absorbed so much infrared to the point where the Kinect sensor could not calculate a depth value for the pixels that were part of it and instead returned null values. The rubber

surface was removed from the table and the depth values could then be calculated. This allowed the surface calibration and detection to work as intended and the difference between the depth values captured with and without the rubber surface are visible in Figure 23.

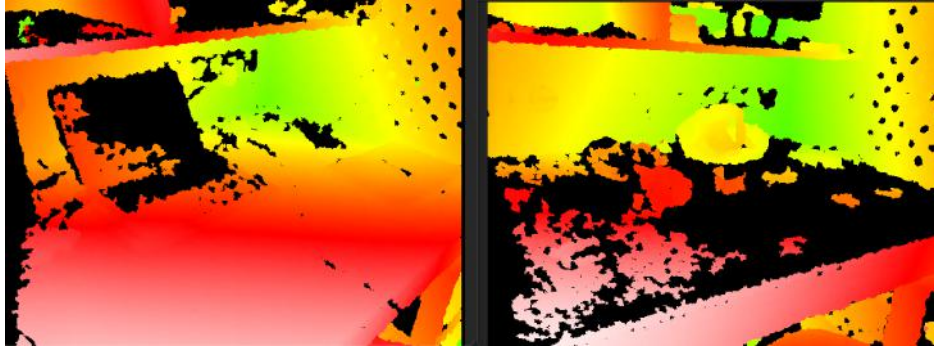


Figure 23.

The difference in the Kinect depth image output for a normal wooden table surface (left) and black matte table surface (right).

The actual surface collision algorithm that is used during the program and determines whether the virtual object is in contact with the surface of the table is straightforward. The coordinates of where the virtual object is proposed to be moved to are provided to the surface collision algorithm. The detected surface array is searched through at the specific proposed coordinates to determine if the table surface is present at or within 15 pixels of those coordinates. If there are no pixels in this region that are part of the surface the algorithm finishes and reports a negative result for a predicted surface collision. If there are pixels in the region that are part of the surface, the depth values of those pixels are compared against the depth values of the proposed virtual object and if they are within 100 depth increments of each other the algorithm returns in the affirmative that a surface collision is about to occur.

3.8 Virtual object implementation and control scheme

In order to insert a 3D virtual object into the environment around the user, a graphical rendering library had to be used. The widely-used 3D graphics library OpenGL [23] supports PC as well as embedded platforms and was chosen as the graphical rendering library used in the system as it supports the design of complicated graphics from low-level graphical primitives such as lines and quads. It is also highly performant due to its low-level and matrix-multiplication-based nature - an essential element of a subsystem of a real-time augmented reality application.

In order to create an augmented reality scene, the virtual object has to be rendered in the same frame as the input image from the Microsoft Kinect camera. Since it is a low-level library, OpenGL supports no background imagery functionality, however the texture of primitive shapes can be set to the value of an array. Reading in the input RGB image from the Kinect and converting it to an array allowed the texture of an OpenGL quad to show the live video input from the Kinect camera. Scaling this quad to the correct size so that its front face filled the window allowed the background to essentially be set to the live video feed.

The "background quad" was rendered in such a way that scaling and rotation transformations performed later on the other virtual object would not affect its location. The virtual object is then rendered as a collection of surfaces and vertices in front of the background quad and with OpenGL depth testing enabled to allow it to be scaled correctly.

OpenGL works with a perspective-based system and not a user-centric camera view system. The camera or viewpoint of the user is always fixed but the OpenGL objects and shapes can be manipulated so that it appears the camera is moving side to side or rotating around an object. The clipping planes of the perspective system define how close or far away objects can be from the origin before they are not rendered and this can be seen in Figure 24. A model-view matrix exists that contains the shapes and textures to be rendered in the output window. Rotations, scaling and translations of objects in the rendered scene are all enacted by manipulating this model-view matrix so that the perspective of the scene appears to change.

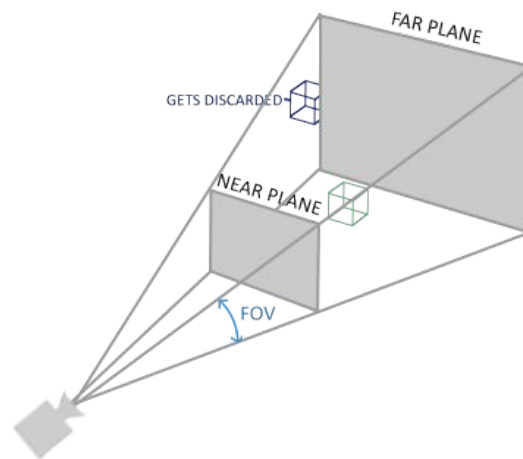


Figure 24.
The perspective system of OpenGL. Reproduced from [33].

The perspective of the system always remains constant and the user or "camera" remains in a static position while perspective "changes" are converted into model-view matrix operations that modify the values there to reflect the desired change in the rendered scene. This allows fast rendering to occur as changes are made with efficient matrix multiplication but presented challenges when trying to model a 3D object and its interactions with two-dimensional (2D) input in the form of a user's hand in the background 2D input image from the Kinect camera. These challenges and the solutions implemented to rectify them are described further below.

The virtual object chosen for the system is a simple cube to accurately reflect the nature of rotations by the user by showing the different faces of the cube in various colours and to simplify interactions with the environment around the user - straight edges make for clear boundaries between objects and their environment. The cube was rendered by creating a list of eight vertices at various coordinates in the OpenGL coordinate system. These coordinates represented the eight corners of the cube and were all normalised to either 1 or -1 along the x, y and z axes to make scaling simple. A list of edges was then created by grouping pairs of vertices to represent a line between those vertices and represent the lines that would form

the sides of the cube. For instance - edge number one was defined as the vertex pair zero and one where the vertices zero and one were (1, -1, -1) and (1, 1, -1) respectively. This pair represented the bottom edge of the front face of the cube.

Two more lists were defined that represented the surfaces of the cube and the various colours to use when rendering each surface. The surfaces list was six items long - one sub-list for each face of the cube, and each sub-list contained groups of the edges that formed part of each surface. Each surface was then rendered using the correct colour allocated to it and each vertex rendered as well. All of this functionality was wrapped in a method so that the cube could be re-drawn at will after rotation, scaling and translation operations had been performed that would change the location, size and orientation of the cube with respect to the background quad and imagery.

In order to move the cube to different locations the model-view matrix could be translated by the amount that the perspective was to be "shifted" and thus the virtual object moved to the side accordingly. This translation would move the cube but the coordinates of the cube would still remain in their normalised form as only the perspective of the user would shift to a new location in order to perform that translation of the cube. This is the core problem of using a projection-based rendering library for a 2D application. A world coordinate system was thus implemented and updated to keep the positions of the virtual object and real-world objects in sync with each other and to enable comparisons between them.

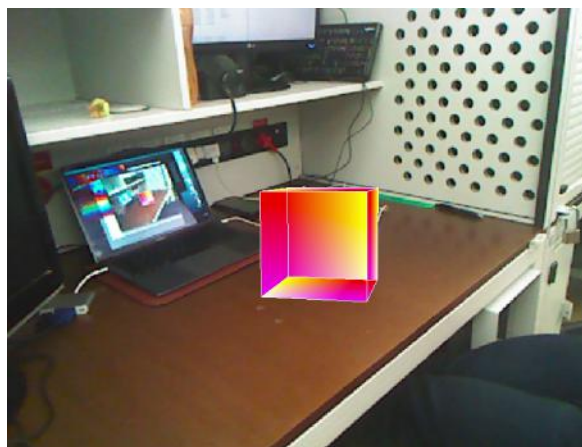


Figure 25.
The starting position of the cube in the OpenGL scene.

At the start of the program the cube was translated so that it was positioned in the centre of the output window as seen in Figure 25. A variable for both the x and y coordinates of the cube were then set with their values being relative to the size of the output window and the amount of unit translations of the cube it would take to move the cube across both the width and height of the window. This meant that in its initial unscaled form, the window was 12 cube translations across from left to right (0-11 translations of 1 unit in the x direction) and 9 cube translations down from top to bottom (0-8 translations of 1 unit in the y direction). Thus the x-coordinate range was 0-11 and the y-coordinate range was 0-8. The coordinate of the cube was set to be the bottom left of the front face of the cube and thus the initial x and y

coordinates were 5.0 and 3.5 respectively.

Now, whenever the cube is translated by a certain value, that value can be normalised to the width and height of the window and the x and y coordinates of the cube updated respectively. Thus a world coordinate system was implemented which allowed the virtual object and real-world objects present in the Kinect imagery to share a coordinate system. This system was compromised however when the cube was either scaled or rotated.

When an OpenGL shape is translated, rotated or scaled the transformation to the model-view matrix is applied at the origin. This is not a problem when no scaling or rotation has taken place as the entire rendered scene shifts to the correct side and the translation appears to work as intended. However, because matrix multiplication is not transient the position of shapes does not remain constant if a rotation or scaling operation is performed on the model-view matrix - all the values are rotated and scaled accordingly. Thus when another translation is performed on the shape the the position of a point is changed and the translation leads to the 2D view of the scene appearing to be incorrect. In order to rotate an OpenGL shape and let the shape retain its position relative to the 2D observer when a new translation is performed, the rotations applied to the shape must all be undone with their inverse, the new translation applied so that the rendered scene is shifted by the correct amount to the side and the rotations all performed again. The result of this process is the rotated cube visible in Figure 26.



Figure 26.
The rotated cube in the OpenGL scene.

To implement this behaviour an array was created that stored all of the rotation angles and the axes they were applied to in the OpenGL scene. This array was reversed each time a new translation was to be made - the rotations applied to the object in their reversed order to return the object to its original unrotated state, the translation performed and then the original list of rotations applied to the object again in order to rotate the object back to its desired orientation.

All of this occurs before the cube is rendered and thus the user sees no additional rotations and the object appears to translate and rotate correctly when handled. This of course leads to a computational increase as the matrix multiplications to be done every time the cube is moved is increased massively but due to the low-level nature of OpenGL and the optimisation

of matrix operations, the additional processing time for this operation is on average only 0.00005197 seconds.

Furthermore, the scaling of the virtual object affects the value the x and y coordinates must be updated with when a virtual object is now translated one cube length across or downwards in the image. The cube when it has been scaled is visible in Figure 27. A variable is used to keep track of the current scale of the cube based on the scaling transformations applied to the model-view matrix and multiplied with the translation value in order to arrive at the correct value the cube is translated with regards to the window x and y coordinates. This process in combination with the list of applied rotations allows the virtual cube to be accurately scaled, rotated and transformed all while maintaining a set of correct and consistent x and y coordinates that shares its axes system with the background input image that itself is projected onto a static quad.

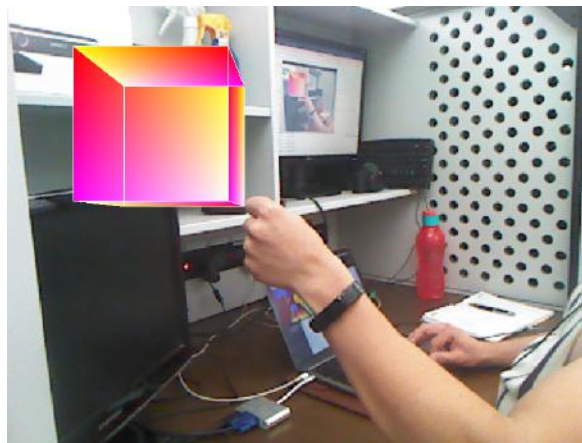


Figure 27.
The scaled cube in the OpenGL scene.

3.9 Object detection and collision avoidance

In order to allow the virtual object to interact realistically with real-world objects in the environment, some form of object detection and collision avoidance system must be implemented. Since only the presence of an object in a particular location relative to the virtual object has to be determined, the simplest method of object detection is to observe the regions on either side of the detected hand region and check if their depth values match the depth value of the hand. In this way objects can be detected that are just to the left, right, top or bottom of the virtual object and are the same distance away from the Microsoft Kinect - right next to each other and if they were both real-world objects unable to move through each other.

While surface estimation had to perform calibration at the start of the program in order to capture a representation of the entire environment around the virtual object including the parts of the surface occluded by the user's hand during operation, object detection can be performed without calibration. In fact, it must be computed repeatedly during the program in order to account for new objects introduced into the scene. Thus an algorithm was developed that

accepted the depth image from the Kinect as well as the coordinates of the hand bounding box which represents the proposed coordinates for where the virtual object should be moved to. The x and y coordinates of the middle of the hand are estimated from the hand bounding box and the depth value of the middle of the hand identified.

The coordinates for a box 10 pixels high and 20 pixels wide is then found that lies just above the hand bounding box and the minimum depth value found for this region - identifying if there are any objects just above the user's hand with a similar depth value to the hand region. The process is repeated for boxes on the left, right and bottom of the hand bounding box. This excludes the side of the hand bounding box that matches the side of the image that the arm enters from - as the depth value of the arm above the wrist will be very similar to that of the hand below the wrist and the false detection of objects around the hand is undesirable. The boxes around the hand bounding box where potential collisions with other objects are detected are visible in white in Figure 28. The location of the boxes had to be modified when the hand tracking algorithm was modified after integration and this made them more accurately spaced around the user's hand as opposed to their previous unreliable locations due to the slightly offset output of the hand detection algorithm.

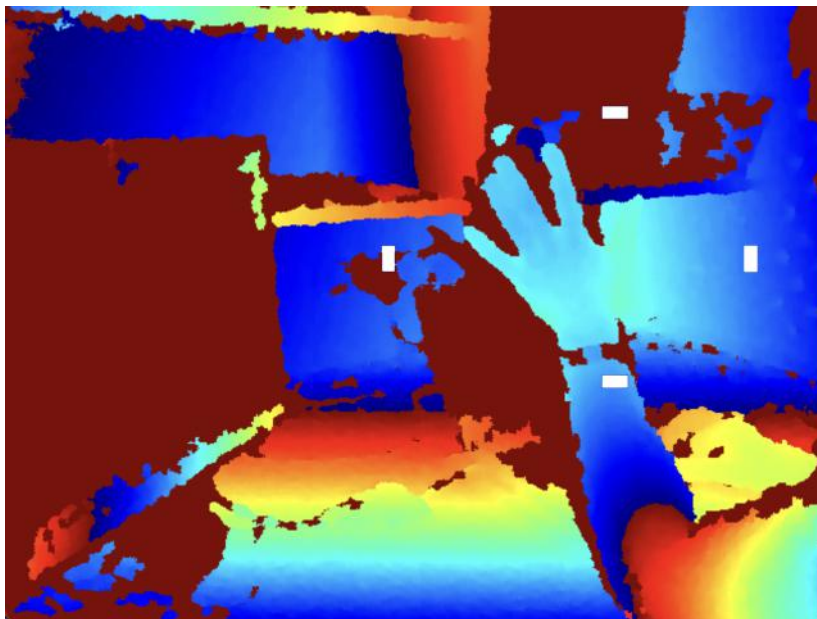


Figure 28.

Depth image showing the bounding boxes in white around the hand where nearby objects are searched for.

Based on the results of the object detection algorithm, a collisions array is created that shows whether a collision will occur if the virtual object is moved to the top, left, bottom or right of its current position. In the main loop of the program, if the hand attempts to move the virtual object in a specific direction, that value in the collisions array is checked before the movement is allowed to occur. Thus object detection and collision avoidance is implemented in a simple and computationally inexpensive manner - a boon to a real-time system.

3.10 Integrated system (main loop)

The operation of the system as a whole takes place in a main loop after all the subsystems have been defined and their functionality encapsulated in methods. The main loop is what is started at the beginning of the program execution. A set of global variables is created to keep track of things like the coordinates of the hand and virtual object scale, coordinates and rotation history. These values are used by many different methods and need to be modified in various places so it makes sense to use global variables. The variables are set to their default values. The main loop comprises of all the main subsystems and its algorithmic description is provided in Algorithm 4.

Low-level OpenGL setup instructions are the first commands executed in the main loop as the OpenGL colour and depth buffers are cleared and a new window for the output display is created. The OpenGL perspective and far and near clipping planes are then set in order to set the "perspective" of the OpenGL window relative to the rendered objects and establish where objects come in and out of view along the x, y and z axes. The initial translation is then performed to move the virtual object to the top left corner and ensure the default x and y coordinates accurately reflect the position of the virtual object.

The surface detection calibration algorithm is then performed if the use of previously captured calibration values is set to false, otherwise those previously captured values are loaded in from a file. If the calibration is performed anew, a window is created that shows the current RGB input from the Kinect and prompts the user to position the camera for the calibration to occur successfully. After the user presses the spacebar the calibration is performed and a prompt shown to the user that it completed successfully - a window is created to show the detected table surface. A while loop is now started that runs for the remainder of the program and is only exited on the termination of the program via the command line or by closing the main Graphical User Interface Graphical User Interface (GUI) window.

At the start of each iteration of the while loop, low-level OpenGL commands are used to reset the model-view matrix back to an identity matrix and disable depth testing. Following this, a frame is read in from both the Kinect's RGB and depth cameras and stored. The RGB image array is converted into bytes and then set to be the texture of a quad that is rendered in the background of the OpenGL scene. When the background has been set, all the intermediate code that could affect the appearance and location of the virtual object is performed. This includes performing hand tracking on the RGB input image to find the location and depth value of the hand according to the detected hand region, classifying the detected hand according to the four different gesture classifiers and then rotating, scaling and moving the cube if the correct gesture is shown. The cube is moved if the delta of the the previous cube coordinates and the proposed new hand coordinates are within 1.5 coordinates of each other and if a fist is recognised by the fist classifier. 1.5 is chosen as the threshold value for grabbing the cube due to the cube coordinate always being linked to the lower left corner of the front face of the cube which can be positioned in various different locations relative to the actual cube based on scale and rotation. Thus to accurately sense proximity to the hand, a slightly larger value is required than being right on top of the coordinate.

Algorithm 4 Main loop

```

1: clear OpenGL buffers
2: set OpenGL perspective
3: perform initial translation to maintain coordinate system
4: if use_preloaded_calibration = true then
5:     load precalibrated depth, normals and detected surface array from file
6: else
7:     desk_in_place = false
8:     while desk_in_place = false do
9:         prompt user to orient camera and calibration rectangle over desk
10:        if user pressed space then
11:            desk_in_place = true
12:        end if
13:    end while
14:    perform surface calibration
15: end if
16: while true do
17:     read in depth and RGB data from Kinect
18:     draw background quad and set texture to Kinect RGB image
19:     perform hand tracking on input RGB image
20:     place extracted hand image into classifier queues
21:     find depth values around detected hand
22:     perform object detection
23:     perform surface detection
24:     if hand present then
25:         retrieve classifier outputs from shared memory variables
26:         if open hand then
27:             perform cube rotation
28:         end if
29:         if fist then
30:             perform cube movement and scaling
31:         end if
32:         display and render the cube
33:     end if
34: end while

```

Additionally, movement of the cube is only allowed if both the surface detection and collision avoidance algorithms return negative collision predictions. The cube is rotated if the gesture for a specific rotation is shown and the hand is open. The rotation takes place and the axis and angle that it is performed with is stored in the rotation history array. Finally, the cube is rendered using OpenGL and the list of vertices, edges, surfaces and colours defined above. Based on the rotations, translations and scaling applied, the cube's position and appearance varies and models realistic interaction with the user and environment. The output of the augmented reality scene is shown to the user in the GUI and feedback regarding surface and object collisions as well as the current frame rate of the system is displayed in the console.

3.11 Multiprocessing

The optimisations described so far were still not enough to achieve the real-time 24fps performance defined by the specifications. Even the introduction of the Numba library for optimising tedious for loops by replacing them with pre-compiled machine code was not enough to achieve the real-time specification and was not suitable for use with original classes like the neural network class. The gesture classifiers were the largest computational bottleneck of the system. The other algorithms such as hand tracking sat idle while the system classified gestures for relatively inordinate amounts of time. Thus the use of multiprocessing was pursued to reduce the total amount of time the program took to run one complete iteration of the main loop. The Python multiprocessing library was used for this.

When parts of a Python program are separated into processes they have separate variable scopes and thus cannot share information normally. Four shared memory variables are thus defined at the start of the program to store the results of the gesture classifier operations. This is in addition to four queues that are used to store the extracted hand images from the hand tracking algorithm and that are used as input to the gesture classifiers. Both these queues and shared memory variables are accessible by all the processes defined in the system. The four gesture classifiers are each placed into their own process and the main loop containing the remainder of the system's algorithms into a final process. At the start of the program these processes are started and their operation continues until the program is halted - executing concurrently.

Additional functions are defined for the gesture classifiers and these are the processes run by the multiprocessing configuration. These functions contain while loops that continuously check if the queue related to that specific gesture classifier is not empty of input images, and if images are available, pops them from the queue and runs the gesture classification operation. The result of which is then placed in the relevant shared memory variable and the loop begun anew. The main loop reads in new Kinect data and performs the hand detection algorithm before storing the result in each of the classifiers' input queues.

The object and surface detection algorithms are then run in the main loop followed by the virtual object rotation and movement operations based on the current value of each of the gesture classifier result variables. It is possible that the algorithms contained within the main loop could be even further split into processes and performance increased further but the difficulty of synchronising these concurrent processes becomes complex and the system already achieves an acceptable performance speed at this point of concurrency, as described below.

The total time taken for all the operations each time a new RGB and depth image is captured from the Kinect camera is shown below in Figure 29. The total time taken includes the processing for hand detection and segmentation, all the gesture classifiers as well as virtual object rendering.

```

Runtime for loop: 0.029920101165771484 s : 33.422346885110045 fps
Runtime for loop: 0.030731916427612305 s : 32.53946112848044 fps
Runtime for loop: 0.03346109390258789 s : 29.885454519544552 fps
Runtime for loop: 0.03366684913635254 s : 29.70280931102125 fps
Runtime for loop: 0.03965902328491211 s : 25.214942708395956 fps
Runtime for loop: 0.04684710502624512 s : 21.346036205220596 fps
Runtime for loop: 0.03413701057434082 s : 29.29371913871254 fps
Runtime for loop: 0.02173900604248047 s : 46.00026321561746 fps
Runtime for loop: 0.030789852142333984 s : 32.47823326261015 fps
Runtime for loop: 0.03220200538635254 s : 31.0539666086699 fps
Runtime for loop: 0.03209114074707031 s : 31.161248142644872 fps
Runtime for loop: 0.031579017639160156 s : 31.66659620088787 fps
Runtime for loop: 0.03260517120361328 s : 30.66998157302056 fps

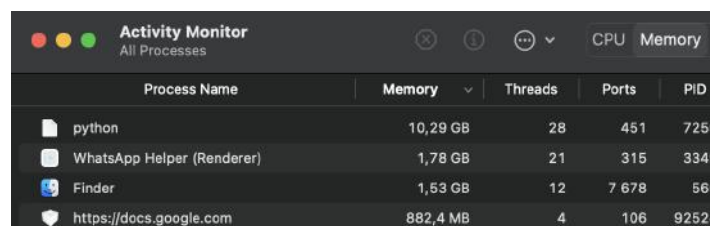
```

Figure 29.
Time taken for each iteration of the main loop

The time per iteration differs based on the amount of computation that had to be done for surface detection, collision avoidance and object detection - all computations that are affected by the position of the virtual object and its proximity to other objects in the frame. Some frames are computed so that the frame rate of the system is as low as 21fps and some as high as 46fps. To account for this, the average frame rate of the system over the previous 100 input images is computed and shown to the user in the console. This value is on average 29.94fps.

One of the main drawbacks of the system is its excessive memory usage. In the process monitoring application of the PC shown in Figure 30 it can be seen that after a few minutes of running the application, the RAM used by the application (labelled python in the diagram) is over 10.29GB. The PC itself only has 8GB of physical RAM and thus swap memory is being utilised to store some parts of the program on the SSD and retrieving them when needed. This gigantic memory usage is hypothesised due to be the storage of the history of rotations applied to the virtual object and used to undo them each time the cube is moved and scaled.

The PC contains a fast SSD and thus the problem is not noticed by the user but could prove to be problematic if the application was used for a long period of time or on a slightly less performant PC. The problem could be alleviated by periodically resetting the object's position to its initial starting point and erasing the rotation history array. Doing this would however break the immersive nature of the application, and is the reason why it was not implemented.



Process Name	Memory	Threads	Ports	PID
python	10,29 GB	28	451	7256
WhatsApp Helper (Renderer)	1,78 GB	21	315	3349
Finder	1,53 GB	12	7 678	566
https://docs.google.com	882,4 MB	4	106	92524

Figure 30.
AR application memory usage

3.12 Embedded platform implementation

Following the algorithmic optimisations and introduction of multiprocessing, the system was finally ported to the embedded platform. The embedded platform chosen was the LattePanda V1 single board computer (SBC) due to its Windows operating system that could easily support VSCode, the Libfreenect Kinect-interfacing library [24] as well as OpenGL. This would make movement of the codebase across platforms essentially seamless. Additionally, the system was low-power and came pre-installed with the OS and three USB ports - exactly what was needed for a keyboard, mouse and Microsoft Kinect to connect to the system. The LattePanda is shown in Figure 31.



Figure 31.
The LattePanda V1 embedded platform.

As the embedded platform was not the original intended operating platform for the system, time was not spent optimising the codebase or interfaces for the under-powered board. Much time was spent attempting to interface the Libfreenect library with the Windows operating system - to no avail. The combination of outdated libraries and lack of support for the Microsoft Kinect V1 in general made it likely that days more troubleshooting, building libraries from source as well as tweaking low-level USB libraries on the LattePanda could be necessary to get the Microsoft Kinect to interface properly.

Thus the decision was made to simply use RGB and depth images captured on the PC and stored in NumPy arrays to provide a set number of input frames to the embedded implementation and not provide live input. This was also because the expected frame rate was so low that real-time use was expected to be impossible anyway. This was expected due to the small amount of memory and processing power available on the embedded platform as opposed to the PC. This is indeed what was observed when the system was implemented on the LattePanda and the frame rate of the system was on average 3.15fps (0.31s per frame) as

seen in Figure 32. The additional time to read in the input images from memory was only 0.02s per image and so was negligible.

```
Runtime for loop: 0.3038755767253222 : 3.290820574579823 fps
Runtime for loop: 0.3040872812271118 : 3.2885295168038815 fps
Runtime for loop: 0.30420157166778067 : 3.287293995614535 fps
Runtime for loop: 0.3063711977523306 : 3.2640143960542805 fps
Runtime for loop: 0.3088093242129764 : 3.238244190160302 fps
Runtime for loop: 0.30954259313562865 : 3.230573181771601 fps
Runtime for loop: 0.30948540488666393 : 3.2311701431161453 fps
Runtime for loop: 0.3103975171738483 : 3.221675254057903 fps
Runtime for loop: 0.3135839240260856 : 3.188938983736981 fps
Runtime for loop: 0.31441890816939505 : 3.1804703025724015 fps
```

Figure 32.
Frame rate of the system implemented on the LattePanda.

The 3.15fps runtime of the embedded system is without multiprocessing. When multiprocessing was attempted to be used on the LattePanda the system ran at roughly 1fps for one or two frames and then crashed. This is hypothesised to be due to the overhead involved in setting up multiprocessing as well as the limited memory bandwidth of the embedded platform - only 2GB of RAM are available on the LattePanda with 32GB of slow solid state storage. Thus swap memory is limited and the already-established high memory requirements of the system are not met.

When multiprocessing was removed, the system could function at the previously-mentioned 3.15fps runtime as the memory overhead of the multiprocessing was removed. Overall the system uses a large number of matrix multiplications for the gesture recognition classifiers and OpenGL model-view transformations. Matrix multiplication benefits greatly from parallelism and explains why the system functions so well on the PC - with the Apple M1 Central Processing Unit (CPU) inside containing four 3.22GHz performance cores and four 2.06GHz efficiency cores alongside 8GB of RAM and ample swap memory available - compared to the LattePanda's Intel Cherry Trail Z8350 CPU with four 1.44GHz cores.

The memory limitations of the embedded platform were again encountered when the system was modified to read in all the static input images at the start of the program instead of reading in each image during the main program operation loop. When this was done, the frame rate of the system dropped to 2.34fps and the LattePanda's RAM was essentially all used and the system became unusable after a few seconds. Thus, the system was reverted to the previous approach.

Thus the embedded application functions as a proof-of-concept but real-time performance is only achieved on the PC implementation due to its superior underlying hardware, better ability to use its multiple cores to handle and take advantage of multiprocessing and overall better computational performance.

4. Results

4.1 Summary of results achieved

Intended outcome	Actual outcome	Location in report
Core mission requirements and specifications		
The system must allow a user to control and manipulate a virtual version of a simple three-dimensional geometric shape object in augmented reality using hand gesture control with less than 41.6ms latency.	The cube could be controlled with hand gestures with an average latency of 33.4ms seconds.	Section 4.2.1.
The system must be able to recognise 9 of a user's discrete hand gestures from video input of a single one of the user's hands in less than 41.6ms	9 gestures can be recognised across four classifiers all with a latency of 33.4ms.	Section 4.2.2.
The system must be able to create and manipulate a virtual object consistent with a real-world object. The virtual cube created must be rotated 360 degrees around the x,y,z axes and moved up, down, left, right, backwards and forwards by 10cm.	The object can be rotated 360 degrees about the x,y,z axes and moved over 50cm up, down, left and right and moved 10cm backwards and forwards.	Section 4.2.3
The system must function in user-apparent real time with no visible latency to the user and the hand model must be updated at 24fps.	The hand model and virtual object function and are updated in real-time with a latency of 33.4ms or at 29.94fps	Section 4.2.1
The virtual object must interact with the real environment it is rendered inside of in a physically realistic and consistent manner. It must remain static for 10s if no input is provided to it, not be able to move through the table surface and be movable only 10cm around objects.	The virtual object could not be moved through the table surface, remained static if no input was provided to it but was only movable 30-40cm around objects.	Section 4.2.4

Intended outcome	Actual outcome	Location in report
Field condition requirements and specifications		
The system should function in a room with at least 300 lux brightness.	The system functioned as expected at 29.94fps under the 371 lux measured in Project Lab 1 and was not tested under different lighting conditions	Section 4.2.1.
The system should accurately classify gestures when only 1 hand is present in the frame and a maximum of 1 finger occluded.	The system could only classify gestures when 1 hand was present in the frame. Multiple hands confused the hand tracking algorithm and led to incorrect classifications, although multiple fingers could be occluded and the correct gesture still classified.	Section 4.2.2.
The system should function when the Kinect, user and virtual object are all within a 2m radius.	The system functioned only when the user's hand was more than 40cm away from the Kinect but up to more than 2m away.	Section 4.2.4.

Table 8.
Summary of results achieved.

4.2 Qualification tests

4.2.1 Qualification Test 1: Test Of Manipulation Of Virtual Object Using Single Hand Gestures In Real-Time

Objectives of the test or experiment

The objective is to confirm that the user can manipulate a virtual object using the discrete gestures of a single hand with a latency of less than 41.6ms.

Equipment Used

- Microsoft Kinect for Windows V1
- Flat table
- Ruler
- Macbook Air M1 PC

Test setup and experimental parameters

The Kinect is connected to the PC and the system initialised. The Kinect is pointed at the environment directly in front of the user and PC with a big enough flat surface visible so that the surface calibration can be successfully performed. Once this is completed, a list of the expected behaviours of the virtual object for the various gesture inputs is generated for comparison against.

Steps followed in the test or experiment

1. The user will display an open hand and move their hand back and forth over the cube for two seconds.
2. The user will make a fist over the cube and attempt to move it to a location 30cm away in the scene.
3. The user will make the gesture for rotation of the cube around one axis and attempt to rotate it in that direction.
4. The observed change in the virtual object will be noted and compared with the expected behaviour for each gesture command.
5. The frame rate counter in the program console will be observed and shown that the frame rate remained above 24fps throughout the different movement of the hand and virtual object.

Results or measurements



Figure 33.

The results of attempting to move the virtual object 30cm across the screen using the closed-fist hand gesture.

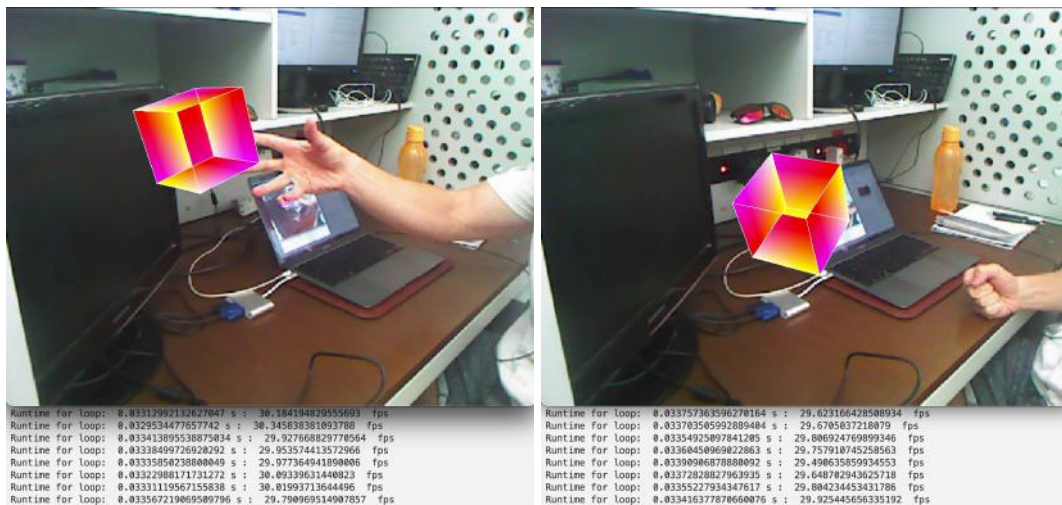


Figure 34.
The program runtime and frame counter being displayed during rotation and movement of the virtual object.



Figure 35.
The user performing the Qualification Test with the physical setup in Project Lab 1.



Figure 36.
The measurement of the brightness in the Project Lab 1.

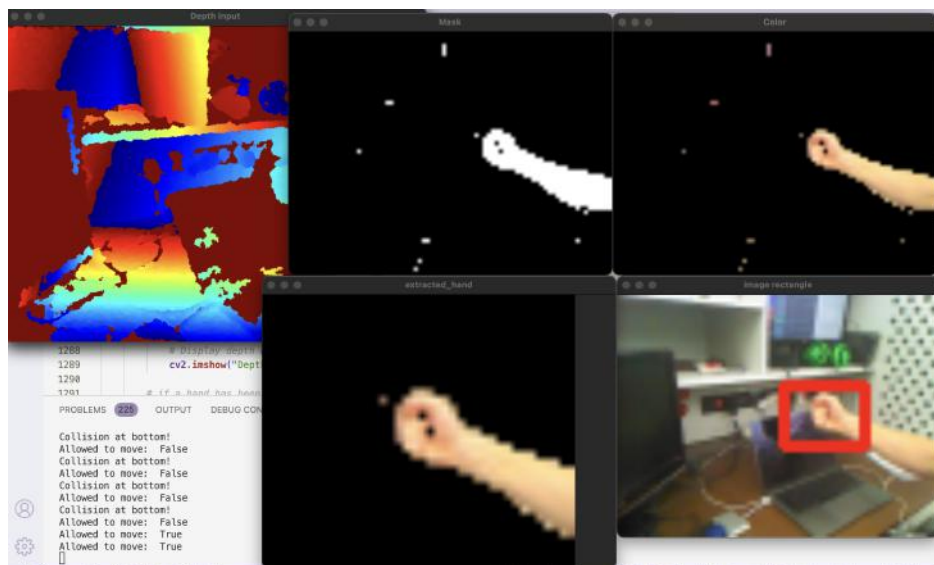


Figure 37.
The output of the other GUI windows showing the YCbCr-segmented image, binary mask and extracted hand region.

Observations

At first, an open hand is held close to the cube as seen in Figure 35 and it does not change position. A fist is then made close to the cube and the hand moved to the right - the cube changes position and follows the position of the hand. In Figure 33 the movement of the

cube more than 30cm is visible by comparing against the ruler in frame. The user then makes rotation gestures with their hand and the execution time of the main loop of the program is printed to the console. Evidence of this is provided in the first part of Figure 34 and shows that the runtime for the main loop of the program is consistently below 0.034 seconds (29fps). Finally, the user makes a fist gesture far away from the cube and as seen in the second part of Figure 34, the execution time of the main loop remains below 0.034 seconds (29fps). Figure 36 shows that the measured brightness in the Project Lab 1 is 371 lux and Figure 37 shows the output of the secondary GUI windows and the output of the background algorithms running in real-time during the qualification test.

Statistical analysis

The average execution runtime for the main loop is calculated by summing the previous 100 runtime values for the main loop and dividing by 100. This allows outlier values to be excluded and is the value displayed in the console. Averaging these values for the runtime values present when rotating and moving the cube in Figure 34 leads to a reliable final average runtime of the main loop of the system as 0.0334 seconds or a frame rate of 29.94 fps.

4.2.2 Qualification Test 2: Test Of Gesture Identification In Real-Time

Objectives of the test or experiment

The objective is to confirm the system can recognise the 9 discrete hand gestures needed to control the virtual object all with a latency of less than 41.6ms.

Equipment Used

- Microsoft Kinect for Windows V1
- Flat table
- Macbook Air M1 PC

Test setup and experimental parameters

The Kinect is connected to the PC and the system initialised. The Kinect is pointed at the environment directly in front of the user and PC with a big enough flat surface visible so that the surface calibration can be successfully performed. Once this is completed, the program is modified to enable text logging so that the currently predicted gesture for the input frame is printed to the console. A small sub-method is activated that will save the prediction of each gesture input to an array.

Steps followed in the test or experiment

1. Modify the program to only allow rotation about one axis at a time.
2. The user displays each of the 11 gestures across the 4 classifiers in various places in the input frame for 1000 input images per gesture.

3. The output accuracy of each gesture is measured.
4. The user displays each of the 3 gestures for 2 seconds each for each of the rotation axes.
5. The console output is recorded and compared against the expected output for that gesture.
6. Modify the program to allow rotation about all axes simultaneously.
7. Display each of the 9 gestures for 2 seconds each.
8. The console output is recorded and compared against the expected output for that gesture.
9. The frame rate counter in the program console will be observed and shown that the frame rate remained above 24fps throughout the different gesture classifications.

Results or measurements

Gesture	Classifier	Prediction Accuracy
Open	Open, Fist	0.836
Fist	Open, Fist	0.756
Down	Down, Side, Up	0.851
Side	Down, Side, Up	0.382
Up	Down, Side, Up	0.987
Down	Down, Forwards, Up	0.881
Forwards	Down, Forwards, Up	0.636
Up	Down, Forwards, Up	0.742
Left	Left, Towards, Right	0.801
Towards	Left, Towards, Right	0.417
Right	Left, Towards, Right	0.689

Table 9.
Real-time accuracy of gesture predictions of system.

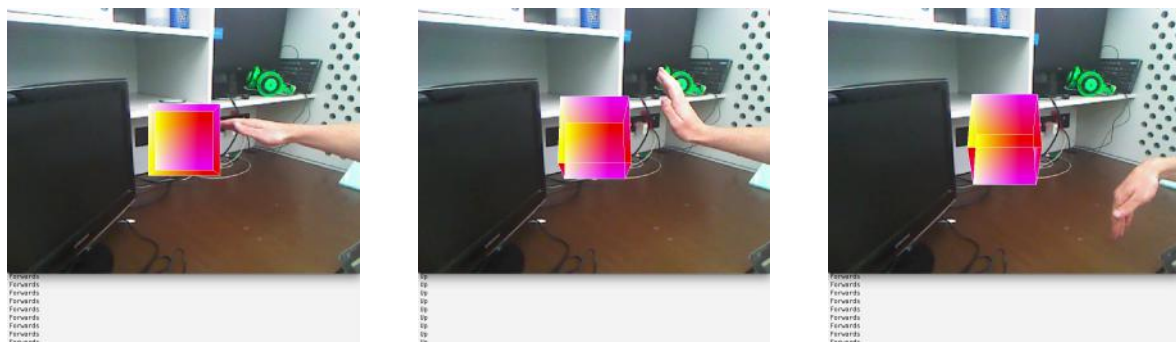


Figure 38.
Output predictions of the Down, Forwards, Up classifier.

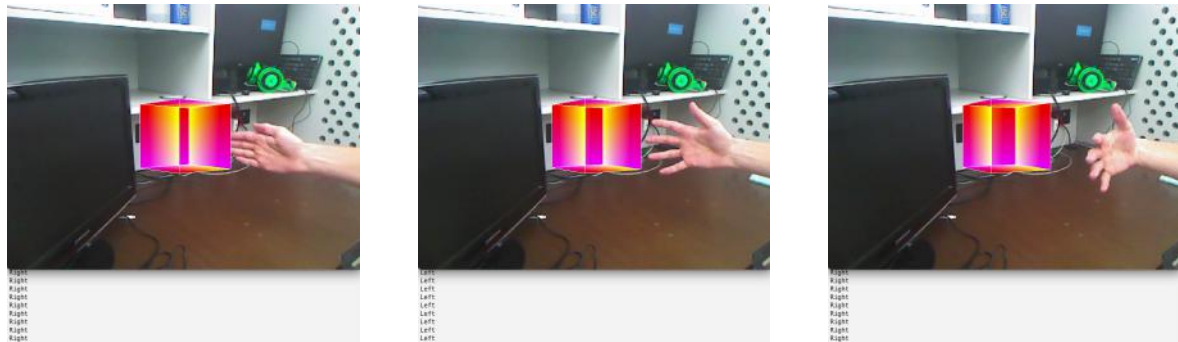


Figure 39.
Output predictions of the Left, Towards Classifier.

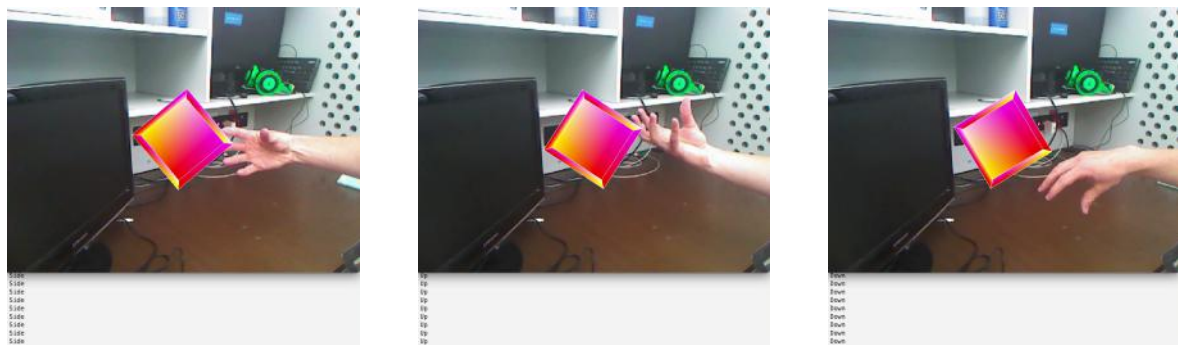


Figure 40.
Output predictions of the Down, Side, Up Classifier.

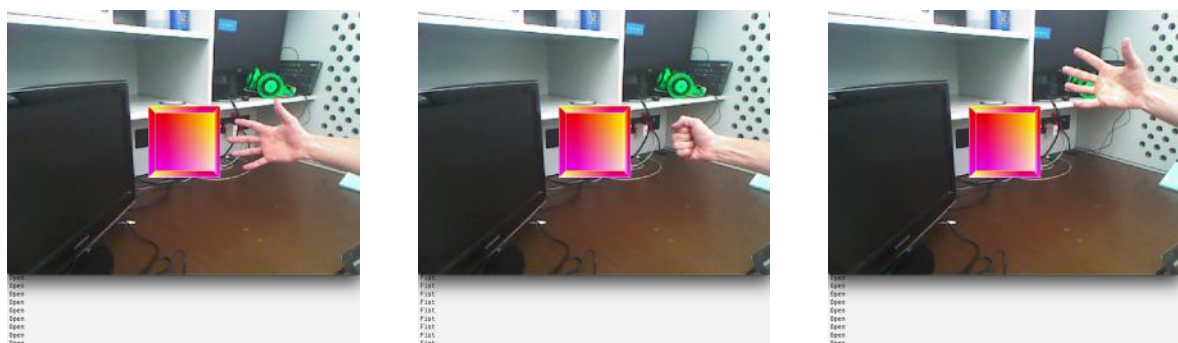


Figure 41.
Output predictions of the Open, Closed Classifier.

Observations

In Figure 38 the user's hand making the forwards, up and down gestures is visible in the input frame and the prediction of the gesture classifier is visible as forwards, up, forwards respectively. In Figure 39 the predictions are shown as right, left, right for the gestures left,

left, towards. The predictions shown in Figure 40 are side, up, down for the gestures side, up, down. Finally, in Figure 41 the predictions are open, fist, open for the gestures open, fist, open. In Table 9 the accuracy of each of the gestures that are a part of the 4 classifiers is visible as they are moved around the input frame by the user and classified by the system - showing the system's accuracy in 1000 instances for each gesture.

4.2.3 Qualification Test 3: Confirming Ability To Manipulate Virtual Object Similarly To A Physical Object

Objectives of the test or experiment

The objective is to ensure that the virtual object can be moved up and down, left and right, backwards and forwards 10cm as well as be rotated 360 degrees using gesture control.

Equipment Used

- Microsoft Kinect for Windows V1
- Flat table
- Ruler
- Macbook Air M1 PC

Test setup and experimental parameters

The Kinect is connected to the PC and the system initialised. The Kinect is pointed at the environment directly in front of the user and PC with a big enough flat surface visible so that the surface calibration can be successfully performed. The ruler is used to determine what 10cm of distance looks like in the scene in front of the user.

Steps followed in the test or experiment

1. The user displays each of the 3 gestures for 2 seconds each for each of the rotation axes.
2. The console output is recorded and compared against the expected output for that gesture.
3. Display each of the 9 gestures for 2 seconds each.
4. The console output is recorded and compared against the expected output for that gesture.
5. The frame rate counter in the program console will be observed and shown that the frame rate remained above 24fps throughout the different gesture classifications.
6. Modify the program to only allow rotation about one axis at a time
7. The user will display the discrete hand gesture for controlling the virtual object's rotation in the positive x direction until the object has rotated in that direction 360 degrees.

8. The user will repeat that process for the rotate positively about the y and z axis commands as well as rotate negatively about the x, y and z axis commands.
9. Modify the program to allow rotation about all axes simultaneously.
10. The user displays each of the 9 rotation gestures for 2 seconds each.
11. The user will hold up their hand and then attempt to move the virtual object 10cm in the scene using the closed-fist gesture and moving the hand to the desired location.
12. The virtual object's position and rotation should be noted at each attempted movement.

Results or measurements

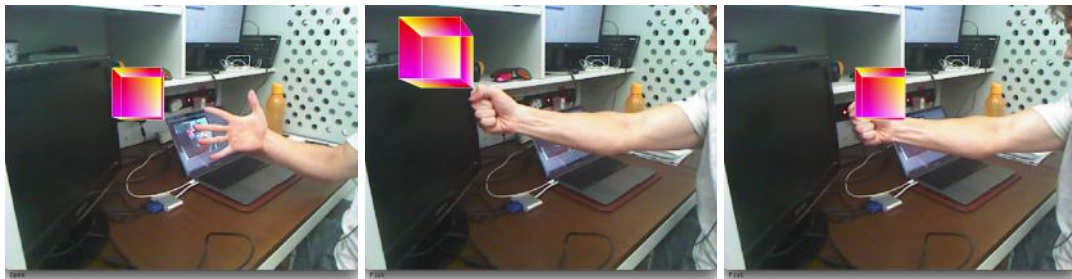
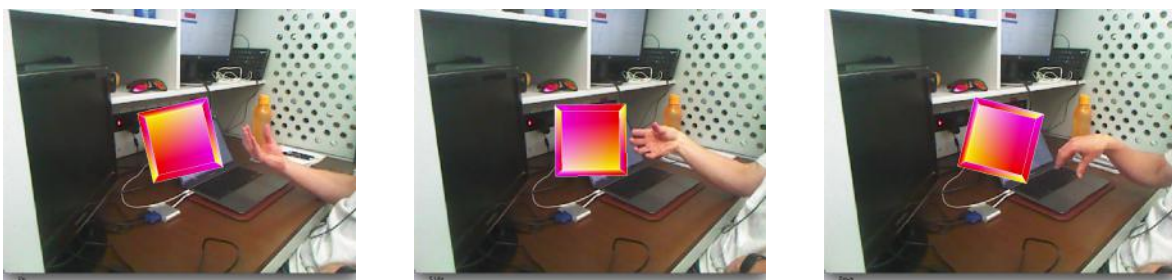


Figure 42.
The results of attempting to move the virtual object around the screen using various hand gestures.



Up

Side

Down

Figure 43.
The results of attempting to move the virtual object about the z axis using the Down, Side Up classifier.



Figure 44.
The results of attempting to move the virtual object about the x axis using the Down, Forwards, Up classifier.

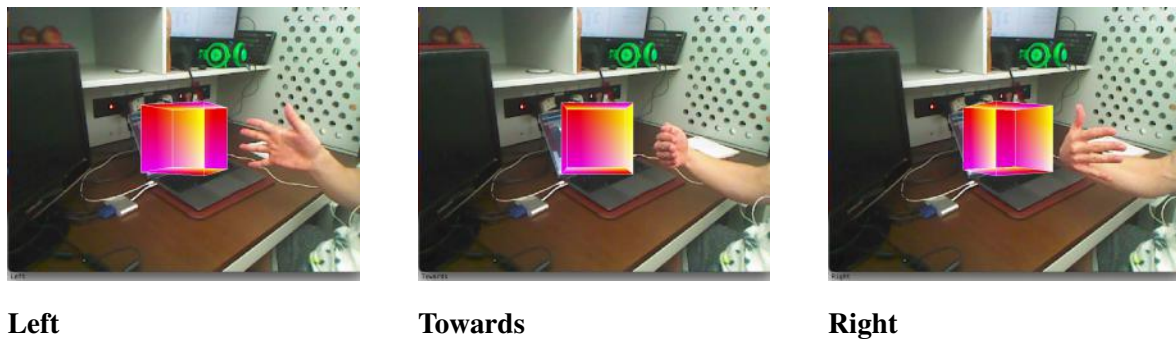


Figure 45.
The results of attempting to move the virtual object about the y axis using the Left, Towards, Right classifier.

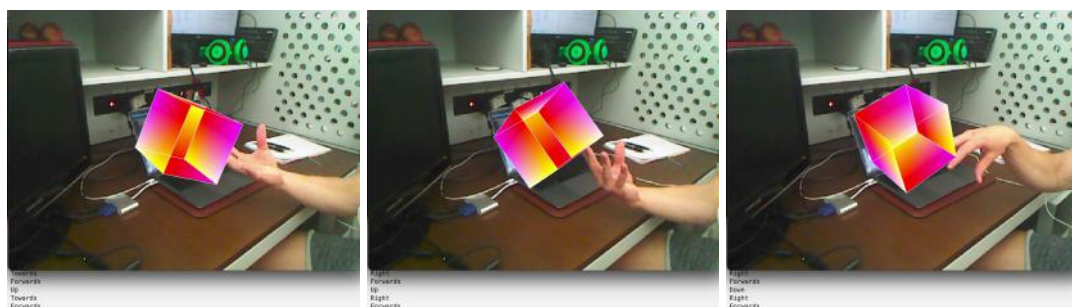


Figure 46.
The results of attempting to rotate the virtual object around multiple axes simultaneously.

Observations

In Figure 42 the movement of the cube is clearly visible when the user makes a fist close to it and then moves their hand down and to the right and the cube follows the location of the user's hand. The changing scale of the cube is also visible between frames as the user's hand comes closer to the camera to grab the cube and retreats when moving it. In Figure 43 the up, side and down gestures are performed and correctly classified by the system - as evident in the console log output of the classifier. The virtual object remains stationary when the side gesture is shown, rotates 1 degree positively about the z axis when the up gesture is shown and 1 degree negatively about the z axis when the down gesture is shown. The object continues rotating in the correct direction by 1 degree per successful classification.

The same process is performed in Figure 44 where the down, forwards and up gestures are performed and correctly classified. The virtual object remains stationary when the forwards gesture is shown, rotates 1 degree negatively about the x axis when the down gesture is shown and 1 degree positively about the x axis when the up gesture is shown. In Figure 45 the towards gesture is shown and the virtual object remains stationary, the left gesture is shown and the object rotates negatively by 1 degree about the y axis and the right gesture is shown and the object rotates positively by 1 degree about the y axis. Finally, the rotation of the cube about multiple axes simultaneously is visible in Figure 46 as the user makes both the up and towards gestures and the cube rotates towards to the user and then makes the down and right gestures in the last frame and the cube rotates downwards and to the right relative to the user.

4.2.4 Qualification Test 4: Confirming Physical Realism Of Virtual Object Behaviour

Objectives of the test or experiment

The objective is to ensure that the virtual object behaves in the physically realistic manner in which a real object would behave, by only being able to move 10cm around real-world objects - not through them, and remaining static on the surface of the table for 10 seconds when left alone.

Equipment Used

- Microsoft Kinect for Windows V1
- Flat table
- Ruler
- Physical box
- Macbook Air M1 PC

Test setup and experimental parameters

The Kinect is connected to the PC and the system initialised. The Kinect is pointed at the environment directly in front of the user and PC with a big enough flat surface visible so that the surface calibration can be successfully performed. The physical box will be placed on a flat surface in front of it. Text logging should be enabled in the program in order to see the result of the surface detection and object collision algorithms.

Steps followed in the test or experiment

1. The system will be initialised so that the virtual cube remains stationary in the image and no input commands will be provided to the system.
2. The user should then attempt to move (using gesture control) the virtual cube past the physical box with a distance of 30cm between the two objects.
3. The user should repeat the movement of the cube with 20cm, 10cm and 0cm of distance in between it and the box.
4. The virtual object's position should be noted at each attempted movement and its distance from the physical box measured by the ruler.
5. The user should attempt to move the cube through the surface of the table by making a fist and dragging the object through the table - its position noted as the user does so.
6. The console should be inspected to see if any collisions with objects or the surface occur while the user performs these tests and if the cube's movement reflects this.

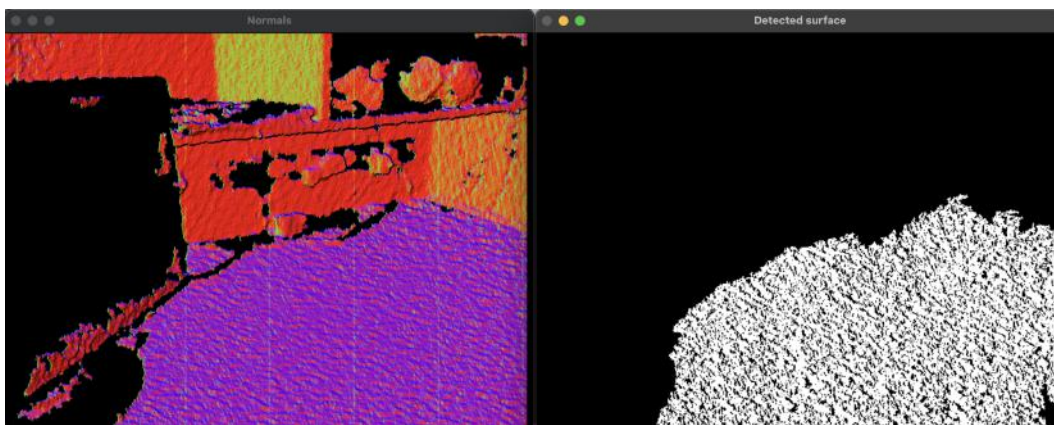
Results or measurements

Figure 47.
The result of the surface calibration algorithm and the extracted table surface compared against in surface detection.

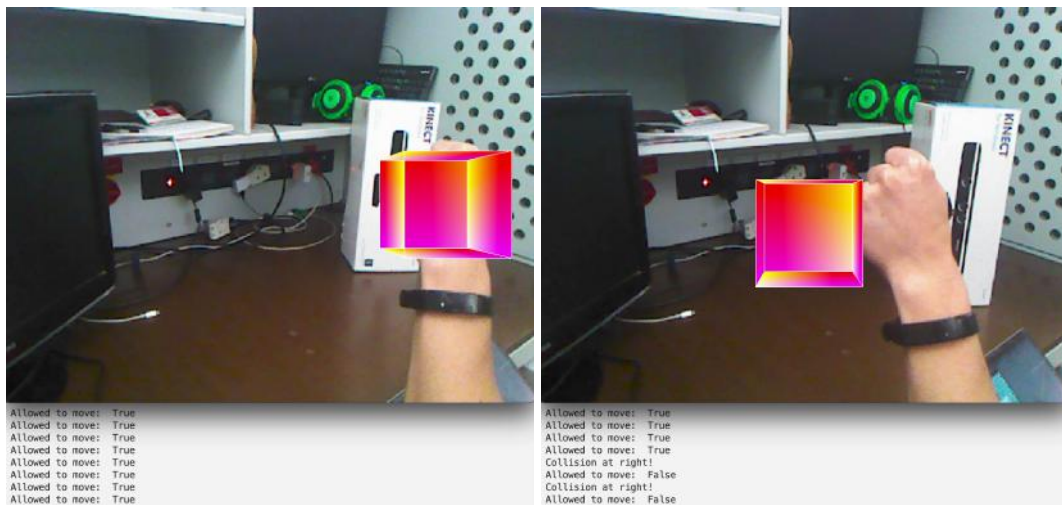


Figure 48.

The results of attempting to move the cube through the box when it is on the right of the user's hand.



Figure 49.

The results of attempting to move the cube through the box when it is on the left of the user's hand.

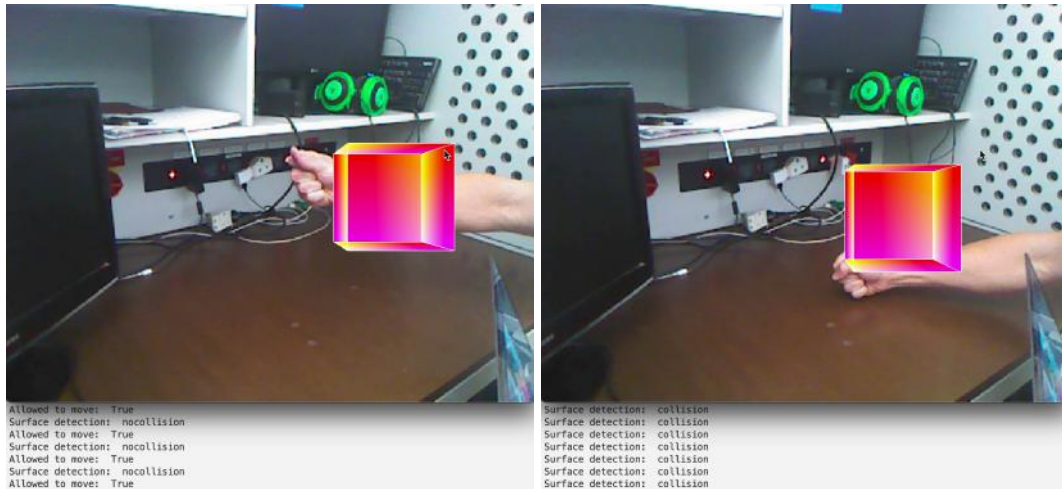


Figure 50.
The results of attempting to move the cube through the surface of the table.

Observations

After the calibration process is performed, the detected surface normals and detected surface are displayed and as seen in Figure 47, the surface of the table has been extracted and is shown in white in the image. When the program has begun, the user attempts to move their hand and the cube through the surface of the table and no collisions are detected when the user's hand is above and away from the surface of the table but a collision with the surface is detected and prevents the cube from moving further. This is visible in Figure 50.

Following this, the physical box is placed on the table 30cm from the user's hand and the user moves their hand right in front of the box. No collisions are detected and the object is allowed to move. The box is then brought forward to just less than 30cm away from the user's hand and a collision is detected - not allowing the cube to move further right. This is all shown in Figure 48 and again in Figure 49 except with the box on the left of the user's hand and a collision being detected not when the user's hand is in front of the box but rather directly in-line with it.

5. Discussion

5.1 Interpretation of results

The most important specification of the overall system was that it runs in real-time or at 24fps. This includes the gesture recognition classifiers as well as virtual object movement and rendering algorithms. Thus the final runtime of the main loop of the program being executed 29.94 times a second is highly desirable and is detailed in Section 4.2.1. This is the most important result of Section 4 and the entire system. It was achieved through a combination of algorithmic optimisation and implementation of multiprocessing to take advantage of multiple CPU cores for running parts of the system in parallel. The hand tracking and gesture recognition classifiers contribute the majority of time to the total processing time and thus the design choices made for them were based on the ultimate goal of real-time 24fps operation. The total time of 0.0334 seconds taken for one run of the main loop is well below the target specification of 0.048 seconds and is achieved in some places at the cost of accuracy.

Being able to manipulate the virtual objects with gestures is the next most important feature of the application. The classifiers built to accomplish this were designed and implemented with the time taken for their computation kept in mind. The results produced in Section 4.2.2 show how when different hand gestures are performed the classifier outputs the predicted gesture at various frequencies of correctness based on the gesture and classifier currently being examined. It is difficult to estimate the accuracy of these classifiers because although the measured accuracies in Table 9 give a good indication of the classifier's ability to correctly predict all instances and permutations of the user's hand that can represent a single gesture - most of these permutations are not used by the user when trying to move or rotate a cube and indeed the accuracy experienced by the user may be much higher than the values presented in this table.

Nevertheless, these objective measurements must be considered and the Open, Fist classifier has an accuracy of 83.6% when predicting instances of an open hand and 75.6% when predicting those of a fist. This shows that the open gesture is much easier to classify. This is likely due to the splayed fingers providing a larger area of pixels for the classifier to recognise an open hand and a closed fist at a further distance from the camera providing a far smaller image overall for the classifier to work with. These accuracies are mostly good enough to reliably grab and let go of the cube but do produce unwanted behaviour when moving the hand or cube to the extreme edges of the image. The Down, Side, Up classifier has accuracies of 85.1%, 38.2% and 98.7% for each gesture respectively and shows that the system cannot identify the side gesture with any reliable accuracy - the point of the user's fingers to the side is clearly misinterpreted as either the up or down gesture instead and is likely due to the system relying too heavily on the pixels above and below the centre of the hand when performing classification - pixels that are missing when the fingers point to the side. The Left, Towards, Right classifier has accuracies of 80.1%, 41.7% and 68.9% respectively. Once again, the towards gesture fails to be reliably interpreted. This means that a rotation about the y and z axes will often be incorrectly induced due to the reasons detailed above.

The Down, Forwards, Up classifier has accuracies of 88.1%, 63.6% and 74.2% when predicting its various gestures and shows that the forwards gesture is harder to classify than the others and that the cube will ultimately be rotated incorrectly more along the x axis than it should be. The accuracy of the classifiers is indeed slightly unreliable due to a gesture in one part of the frame not always being correctly predicted as when it is in another part of the frame. A large amount of training data was provided to the classifiers with but incorrect predictions are still caused due to what is hypothesised to be garbage-in-garbage-out.

This is because the input image from the hand tracking algorithm is resized when provided to the classifiers in order to reduce their computational load and the amount of pixels considered during classification. In this resizing process, some of the fine-grained details of the hand gesture are lost to blur and noise. The resizing was maintained due to the trade-off of speed versus accuracy but sometimes led to incorrect output predictions being provided by the classifiers. Overall, the results of the gesture classifiers are acceptable for the rough control of a virtual object but the accuracy is deficient for fine-grained manipulation.

The results of the system's ability to translate the user's gestures into the correct rotation and movement of the virtual object are, however, very good. The specification of being able to rotate the cube about all axes 360 degrees and move the cube 10cm in all directions is met and exceeded. When the correct gesture is displayed the cube is rotated 1 degree about the correct axis for every predicted gesture and continues rotating indefinitely. The changes in gesture recognition predictions can make the rotation of the virtual object jerky though and reduces the immersion of the augmented reality application. When the user makes a fist and "grabs" the cube to drag it to a new location, the cube is moved accordingly. This is all accomplished within the latency requirement of 0.048 seconds. The virtual object can in fact be moved over 50cm in all directions except the backwards and forwards ones - where the discontinuities or noise in measurements from the Kinect's depth sensor can cause incorrect scaling to be applied.

The operation of the surface calibration algorithm is highly successful and outputs a near-total mask of the table surface in the input image. This mask is then used to detect collisions between the virtual object and the surface of the table. The test conducted in Section 4.2.4 shows that the virtual object is moved downwards with no surface collisions detected but as soon as the user's hand comes within a 10cm range of the surface of the table, a collision is reported and the object's movement downwards stopped. The surface detection algorithm however fails to accurately discriminate between an object that is far in front of the surface table in the z-dimension (closer or further away from the camera) and thus does not always accurately detect collisions based on the depth value of the hand. This is difficult to examine however due to other parts of the table being closer to the user in the frame and the requirement that the camera be within a 2m radius of both the user's hand and the table. The virtual object does remain static on the table when no gestures are provided to it though and so overall, the surface detection algorithm is successful in meeting specifications.

The remainder of Section 4.2.4 outlines how the object detection algorithm successfully detects objects placed in the way of the the user's hand moving the virtual object. The distances between the virtual object and real-world box are however not to specification and the collision cannot be detected only 10cm away - the threshold of the object detection must be set to detect objects 30cm away in order to not miss collisions. Thus the object

detection algorithm functions but without the accuracy and reliability outlined in the original specification.

Considering the approach to the system's design outlined in Section 2, the initial selection of the Kinect sensor as the input device was based on the desire to avoid expensive computation to acquire depth information for use in surface and object detection. This was achieved by using the Kinect and the RANSAC algorithm for efficient plane and surface detection as well as an original algorithm for object detection.

The use of alternative larger gesture recognition classifiers was considered but ultimately rejected in favour of smaller ones that could run much faster and lead to a real-time system - this was ultimately successful and validated the approach in Section 2. The approach of using CNNs over classical approaches like fingertip contour estimation was also validated by the ability to classify multiple gestures with acceptable accuracy and not have to design many small-scale detectors and algorithms for differentiating gestures based on features like hand size and fingertip location - this was performed and learned by the convolution step of the CNN. The use of OpenGL and its low-level primitives to create a fast rendering virtual object was validated too due to the ultimate real-time operation of the system at 29.94fps as opposed to another heavy rendering tool that would have been a computational burden.

The decision to use Python to develop the system was made with the expectation of much development and testing when using machine learning approaches and the relative development ease of using a dynamically-typed interpreted language. Additionally, the large collection of optimised algorithms available in the NumPy library for linear algebra and matrix computations increased performance to the point of comparison with a lower-level language anyway. This is because these functions are written in the lower-level language C and merely interfaced with using a Python wrapper, making the re-writing of them from first-principles in C an unnecessary task that was not the main goal of the system.

The ultimate real-time performance of the system validates this approach and the design changes discussed in Section 3 show how the iterative development process benefits from it. The use of TensorFlow is similarly validated by the graph searches undertaken in Section 3.6 that would have taken a number of weeks using the first-principles backpropagation implementation but could all be performed in a single day using TensorFlow.

Ultimately, the system does solve the problem of an augmented reality virtual object gesture control application as it allows interaction with a virtual object similar to the application built by Billingham [1] for treating arachnophobia but with simpler and different sub-components. The system makes use of simple CNNs inspired by Fan [13] but with the addition of classical colour segmentation approaches such as those used by Jones. [34] The real-time operation of the system and its lightweight computational footprint comes at the expense of accuracy and would be ideal for home-use applications or being added on top of existing applications in order to provide gesture control.

5.2 Critical evaluation of the design

Taking the interpretations of the results provided above as well as the approach to designing the system and work done by others in the literature all together, a critical evaluation of the

designed system can now be provided.

5.2.1 Aspects to be improved in the present design

The largest improvement to the current efficacy of system would be to improve the accuracy of the gesture classifiers - a higher real-time accuracy for predictions would reduce spurious rotations during the operation of the system and prevent the cube from being "dropped" during movement. This could be accomplished by training the existing classifiers with even more numerous sets of clearly well-defined and distinctive training images in addition to modifying the hand tracking algorithm to output higher quality extracted images of the hand.

Alternatively, different classification models could be used that would perform the task of gesture classification more adeptly. This could be in the form of much larger models that use intermediate transforms to predict hand landmarks such as Mediapipe [12] but would need to be heavily modified and optimised in order to run in real-time using a first-principles implementation. Additionally, the gesture classifiers sometimes failed to correctly identify the same gestures that were shown by a hand close to the camera and one far away - introducing intermediate classifiers to distinguish whether a hand is far or close from the camera and then using a separately-trained model accordingly to classify that gesture could be a potential way to increase gesture recognition accuracy.

It was discovered too after the final integration of the sub-components of the system that the gesture classifier need not be run as often as it currently is. By reducing the amount of times the classifier is run from 29.94 times a second to only 5, the accuracy of gesture predictions even for fast-moving objects would not be greatly reduced but the computational load on the system would be and would allow the gesture recognition classifier used to be much larger and more computationally expensive - gaining accuracy without trading off speed. Using a different classifier that outputs a more detailed model of the hand would offer many more degrees of freedom to the virtual object control mechanism - allowing the user much more fine-grained manipulation.

The overall augmented reality integration of the 2D input image with the 3D virtual object leaves much to be desired. Movement of the virtual object along the x and y dimensions is accurate and realistic but the scaling in the z dimension is unreliable and inconsistent at times depending on the Kinect depth data. Improving this so that the virtual object interacts with objects based on their depth value more accurately and realistically would yield a more immersive augmented reality experience.

Of particular note is the weak performance of the system on the embedded platform - the LattePanda V1. The implementation is not interfaced with the Kinect due to library installation issues and the implementation only runs at 3.15fps - a far cry from real-time, when fed in video and depth input captured ahead of time on the PC. Optimising algorithms and parts of the system for use on an embedded platform would be necessary as well as even potentially swapping out sub-components of the system like the gesture recognition classifiers for simpler non-machine learning techniques like the classical contour-finding algorithm that are not as robust or reliable but would require less computation.

The choice of the LattePanda as the embedded processing platform appeared theoretically sound as it possessed similar specifications and processing power to other embedded platforms at the same R3000 price range of the Project budget. However, choosing hardware with better support for the libraries needed for interfacing with the Kinect and a system with a less resource-intensive operating system would benefit the performance of the embedded version of the system greatly. As it stands currently, it is more of a proof of concept than a usable real-time system.

The object detection algorithm has the drawback of only being effective when the threshold for detecting objects is larger. This causes objects to be detected in a certain direction even if the object is further than 30cm behind or in front of the user's hand. Making this algorithm more accurate and reducing the amount of false positive collisions detected would be advantageous and allow better integration between the virtual and real-world objects. The same holds true for the surface detection algorithm and including the effect of the depth value more when determining if the virtual object is not only at the correct x and y coordinates of the surface of the table but also at an appropriate z coordinate.

The use of multiprocessing introduced a new challenge to the system in that the classification of gestures was much slower than reading in input images and performing hand detection and thus necessitated the use of queues to store input images. Synchronising the reading of these queues with the output of the gesture classifier so that the current gesture made by the user is in-line with the current prediction required popping multiple items off of the queue at once when determining a new classification. Overall, the introduction of multiprocessing introduced more difficulty in managing the state of variables and the system when simultaneous execution was performed - this could be improved and streamlined to reduce the lack of synchronisation.

Finally, the use of OpenGL for the virtual object rendering necessitated the storage of every rotation applied to the virtual object in order to undo them each time the object is scaled or moved and then re-apply them afterwards. This ensures the rotations and translations keep the virtual object's position and rotation correct and does not impact the system's speed in any appreciable way but it does become a large memory drain as the system runs - storing all the rotations made throughout the program's runtime. Occasionally resetting the virtual object's rotation or finding a more efficient way to store these past rotations would be beneficial to the memory footprint of the application.

5.2.2 Strong points of the current design

The biggest advantage of the system is that it runs at above real-time performance - 29.94fps. This is due to the multiprocessing and algorithmic optimisation implemented as well as choice of lightweight gesture classifiers. These classifiers achieved the goal of gesture recognition to a large degree while remaining small and not requiring as much computation as some larger models in the literature. The training of these classifiers could be achieved on a laptop and the models were easy to swap in and out of the integrated system thanks to a modular class-based design and loading of the classifier weights.

The use of multiprocessing sped up the system significantly and reduced bottlenecks that occurred when classification had to wait for input imagery to be provided to the system and

vice versa. It allowed slightly larger gesture classification models to be used and less optimised versions of the hand tracking and surface detection algorithms to run in the background and not impact overall performance as they were running in their own processes.

The hand tracking algorithm itself is a strong point of the system as it does not require a heavy machine learning algorithm to detect and segment a hand from the background but rather makes use of efficient classical techniques like colour segmentation and region summation. It is robust against occluded fingers and different hand colours thanks to the hue segmentation performed with the YCbCr colour space and doubles as the real world objects' interface with the world coordinate system.

The object and surface detection algorithms are both lightweight and make good use of the Kinect input data to reduce computations and still perform their functions quickly. The surface detection algorithm in particular gives a very fine-grained model of the user's environment and could easily be extended to find all planar surfaces in the input image - allowing more object detection and surface interactions if desired.

5.2.3 Under which circumstances is the system expected to fail

The hand tracking algorithm will not function correctly when multiple hands or other skin-coloured objects like faces are present in the input image - the hand detector will get confused and mistakenly identify the face or other arm or part of hand as the detected hand. It will be a graceful failure as the location of the hand will seemingly flit erratically about the input image and induce spurious rotations of the virtual object but the program will not crash.

Complicated hand gestures like a half-open fist will confuse the gesture classifiers and induce incorrect gesture predictions. This is due once again to the classifier's tradeoff of speed versus accuracy and due to the difficulty of detecting between a loosely held fist and an open hand that has stiff fingers. Additionally, making gestures near the extreme edge of the frame can lead to warped extracted images from the hand detection algorithm and thus unreliable gesture classifications.

The object detection algorithm is expected to fail when objects are very thin or small so that their depth value cannot be easily found. It is also expected to fail when objects are within 40cm of the user's hand - false collisions can be produced at this range due to the large region the algorithm searches for collisions in order to not miss any.

The system is also unreliable when the environment in front of the Kinect contains many matte black surfaces as the Kinect's infrared sensor is unable to reflect infrared light off these surfaces properly and gauge their distance value correctly.

5.3 Design ergonomics

The entire system is premised on the assumption that augmented reality can provide intuitive and natural interactions with digital interfaces and virtual objects. The control of the virtual object is performed using various gestures that are similar to the same gestures the user would use when manipulating a physical object in the real world. These include grabbing the virtual

object by closing a hand around it, moving it by maintaining the closed hand and moving the hand to a new location as well as rotating it by twisting the wrist and hand. These are all intuitive gestures that do not require physical hardware and allow the user's body to become the control interface with the system - the epitome of ergonomics.

The system also provides a simple GUI with no misleading icons or text and simply shows the user the scene in front of them with the rendered virtual object. More detailed information is shown to the user in the console such as notifications about collisions with the surface or other objects.

Because the system is very software-focused and only has two hardware components - the Microsoft Kinect and a PC, the setup of the system is very simple and is no more complicated than plugging the Kinect into power and the PC. The system does require calibration at the start of the program but this is a simple process that shows the user a rectangle and asks them to move the camera until the surface of the table lies underneath that rectangle. That is the only additional setup required for the system and because so much of the complexity is abstracted away by the software, it is a simple system for new users to use.

5.4 Health, safety and environmental impact

The Microsoft Kinect camera is mounted on top of a tripod and might be easily bumped off and onto the ground or user if the user bumps it or sits while using the system and the side of their chair knocks into the tripod. Thus the use of copious amounts of Prestik to secure the Kinect to the tripod ensures that small bumps or jostling of the tripod does not cause damage to the Kinect or allow it to fall onto the user and harm them.

The system contributes towards greenhouse emissions by using electricity however the embedded system is a system-on-a-chip and thus uses very low amounts of power to run. This is true too for the Macbook Air M1 laptop used as the PC - the Apple Silicon CPU architecture is known for its power efficiency. The system was designed overall to interface simply with an existing PC and thus does not require new hardware besides the Kinect to be manufactured, transported or installed by the user that might adversely affect the environment. Additionally, the Microsoft Kinect was originally designed as a video game input device but found new life within academic use cases and so has the advantage of being a repurposed design and device that did not require brand new technology or emissions to take place to develop it.

The Kinect can be recycled at an e-waste facility one day when it is no longer usable and the PC donated to a charitable cause or also to an e-waste facility. Finally, the entire system is a demonstration of a potential future technology - gesture input to digital devices, that might one day reduce the amount of external peripherals developed in order to interface with modern computing systems. This is a net positive for the environment as it will decrease the amount of physical resources used up in technology production and in transporting those goods across the world to consumers.

5.5 Social and legal impact of the design

Few legal restrictions exist regarding the type of input devices allowed for digital systems but whenever a computer system interacts with a person, the Protection of Personal Information (POPI) Act must be considered and how the system respects the user's right to privacy and the protection of their personal details. Facial recognition and the tracking of users has become an issue recently but the system adhered to the POPI Act by not storing any information about the user, no information is sent off of the user's device and indeed due to the way the Kinect is set up to point to the desk and have the user "reach into" the scene, the user's face is not visible in the frame and since a way has not yet been developed to identify people based on their hands from an RGB image, the system respects and preserves the user's privacy.

As for the social and cultural influence of the design, the system has the potential to make computing and digital technology more accessible to uneducated people or those that have no prior experience with digital interfaces. This is because gesture recognition is available to almost everyone and the intuitive gestures that a user uses in real life can be extended to interacting with virtual objects and augmented reality interfaces. The advent of gesture recognition shown by this system might change the way people interact with digital systems in future, saving money spent developing complicated user input devices and reducing the time spent learning how to interact with new systems.

6. Conclusion

6.1 Summary of the work completed

This report describes the work done to implement a real-time augmented reality application that uses gesture control to manipulate a virtual object. The system had to operate in real-time at 24fps and allow the user to rotate and move the virtual object in all directions in a realistic manner.

A literature survey was conducted and outlined the previous work done in gesture recognition, augmented reality applications, surface detection and the sub-components necessary to build out and integrate these systems. The design and implementation of these subsystems was then completed from first principles. The main components of the system were the gesture recognition, virtual object control, environment recognition and augmented reality creation subsystems. The Microsoft Kinect sensor was integrated with a PC and the software developed for the system. The design of a first-principles CNN architecture was undertaken and used alongside TensorFlow to train a number of separate gesture classifiers.

The virtual object control scheme was interfaced with the gesture classifiers to allow movement and rotation instructions for the virtual object to be created. The environment recognition subsystem was developed using a modified RANSAC plane-estimation algorithm as well as an object detection algorithm that both relied on the Microsoft Kinect depth data to detect collisions of the virtual object with the surface of a table and objects around the user. Finally, the augmented reality creation subsystem was developed with OpenGL and integrated with the gesture classifiers to provide the user the ability to rotate and move the virtual object with gesture input as desired.

All the software was developed in Python and optimised for speed of operation. The system was implemented on both a Macbook Air M1 PC and a LattePanda V1 embedded SBC. Several qualifications tests were performed to validate the design and implementation of the system according to the design specifications. The most important result of the system is shown in Section 4.2.1 and shows that gesture control of a virtual object is possible with a latency of less than 0.0334 seconds or over 29.94fps of real-time operation.

6.2 Summary of the observations and findings

Successful gesture control of a virtual object could be accomplished in real-time at a frame rate of 29.94fps. All of the underlying subsystems could function within the 0.0334 seconds latency requirement. The gesture recognition subsystem allowed the successful detection of the 9 different gestures with good theoretical accuracy and acceptable accuracy during real-time operation. The virtual object could be successfully rotated 360 degrees about the x, y and z axes and successfully moved over 50cm in the up, down, left and right directions and 10cm in the backwards and forwards directions.

Realistic behaviour of the virtual object as similar to that of a real-world object is achieved

through the implementation of a surface detection algorithm that prevents the virtual object from moving through the surface of a table by detecting the surface successfully from 10cm away during movement of the virtual object. Additionally, the implementation of an object detection algorithm prevents collisions between the virtual object and real-world objects surrounding the user - with below-specification accuracy.

Ultimately, the system was a success in that the virtual object could be controlled with gesture input at the required real-time speed. However, this speed was a direct tradeoff of accuracy as the gesture recognition subsystem left much to be desired when classifying complex or transitive gestures. This tradeoff was made intentionally alongside the use of multiprocessing, a computationally simple hand tracking algorithm and effective use of the depth data of the Microsoft Kinect for object and surface detection to ultimately all achieve a real-time implementation of the system.

The embedded platform implementation of the system was not integrated with the Microsoft Kinect and did not meet the real-time speed specification - only running at 3.15fps with previously captured input fed into it. It exists as a proof of concept to complement the fully-working and specifications-fulfilling PC implementation. It was also realised during testing of the system that the need for real-time operation does not mean that all subsystems have to operate at this speed and that running the gesture recognition subsystem at only 5fps would still predict gestures accurately even for a fast moving hand, but the system would be relieved of a lot of computational burden which could be directed towards a larger and more complex gesture classifier architecture that could increase the accuracy of the virtual object control and manipulation greatly.

6.3 Contribution

Some of the new theoretical background that had to be obtained to complete the project included details about RGB and YCbCr colour theory, the convolutional part of a CNN and the mathematics of forward propagation and backpropagation through a CNN. New hardware mastered for the implementation of the system included the Microsoft Kinect and understanding how its depth data is captured and represented in matrix form. New software utilised in the development of the subsystems included OpenGL for graphical 3D rendering and TensorFlow for training neural networks.

Many novel algorithms were developed for the implementation of the system. These include a first principles hand detection and tracking algorithm as well as novel surface and object detection algorithms that relied upon the Kinect depth data. Furthermore, a first-principles CNN was developed for gesture classification that took inspiration from the Keras layers API and a surface calibration algorithm that made use of a modified RANSAC-inspired plane-estimation algorithm. While these algorithms were not completely novel they were designed from first principles to be modified to suit the needs of the system.

The system was developed under the supervision of the study leader who provided guidance on focusing on literature and not merely hacking together naive approaches and algorithms and additionally provided training advice regarding neural networks and suggestions to investigate data augmentation amongst other approaches to increasing neural network prediction accuracy.

Meetings twice a month kept focus on the overall system and integration and the student was dissuaded from obsessing over small details of subsystems that didn't affect the overall performance or final implementation.

The code developed from first principles was described above but further use of libraries was made to complete the system's sub-components. This included using TensorFlow for neural network training, Libfreenect for interfacing with the Kinect, NumPy for linear algebra and array operations, OpenCV and Pillow for some image processing, Multiprocessing for running code in parallel and OpenGL for graphical primitives rendering.

6.4 Future work

The biggest deficiency of the system was the accuracy of the gesture recognition classifier and thus improving this accuracy should be the most prominent focus of future work. Choosing a better more complicated classifier that uses more computation can be offset by running the classifiers less often while still retaining accurate predictions of fast moving users.

A better integration of the world coordinate system and virtual and real-world objects could be implemented with a deeper focus on the depth information and z-dimension of the system to increase the realism of the augmented reality application. This deeper focus on the z-dimension could be implemented in the surface and object detection systems too to increase detection accuracy and allow more fine-grained collision detections to occur.

Finally, the selection of a different embedded platform that has better support for depth-sensing hardware, more processing power and a less resource-heavy operating system would improve the system's embedded performance greatly.

7. References

- [1] M. Billinghurst, T. Piumsomboon, and H. Bai, “Hands in Space: Gesture Interaction with Augmented-Reality Interfaces,” *IEEE Computer Graphics and Applications*, vol. 34, no. 1, pp. 77–80, 2014.
- [2] M. Baldauf, S. Zambanini, P. Fröhlich, and P. Reichl, “Markerless visual fingertip detection for natural mobile device interaction,” in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011, pp. 539–544.
- [3] H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana, “Virtual object manipulation on a table-top AR environment,” in *Proceedings IEEE and ACM International Symposium on Augmented Reality (ISAR 2000)*. Ieee, 2000, pp. 111–119.
- [4] V. Buchmann, S. Violich, M. Billinghurst, and A. Cockburn, “FingARtips: Gesture Based Direct Manipulation in Augmented Reality,” in *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 2004, pp. 212–221.
- [5] R. Schnabel, R. Wahl, and R. Klein, “Efficient RANSAC for Point-Cloud Shape Detection,” in *Computer graphics forum*, vol. 26, no. 2. Wiley Online Library, 2007, pp. 214–226.
- [6] B. Nuernberger, E. Ofek, H. Benko, and A. D. Wilson, “Snaptoreality: Aligning Augmented Reality to the Real World,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 1233–1244.
- [7] C. Liu, K. Kim, J. Gu, Y. Furukawa, and J. Kautz, “Planercnn: 3D Plane Detection and Reconstruction From a Single Image,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4450–4459.
- [8] M. Y. Yang and W. Förstner, “Plane Detection in Point Cloud Data,” in *Proceedings of the 2nd int conf on machine control guidance, Bonn*, vol. 1, 2010, pp. 95–104.
- [9] H. F. T. Ahmed, H. Ahmad, K. Narasingamurthi, H. Harkat, and S. K. Phang, “DF-WiSLR: Device-Free Wi-Fi-based Sign Language Recognition,” *Pervasive and Mobile Computing*, vol. 69, p. 101289, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1574119220301267>
- [10] K. Sadeddine, F. Z. Chelali, R. Djeradi, A. Djeradi, and S. Benabderrahmane, “Recognition of User-Dependent and Independent Static Hand Gestures: Application to Sign Language,” *Journal of Visual Communication and Image Representation*, vol. 79, p. 103193, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1047320321001231>
- [11] Y. Shen, S.-K. Ong, and A. Y. Nee, “Vision-Based Hand Interaction in Augmented Reality Environment,” *Intl. Journal of Human–Computer Interaction*, vol. 27, no. 6, pp. 523–544, 2011.

- [12] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, C.-L. Chang, and M. Grundmann, “MediaPipe Hands: On-device Real-time Hand Tracking,” 2020. [Online]. Available: <https://arxiv.org/abs/2006.10214>
- [13] Q. Fan, X. Shen, Y. Hu, and C. Yu, “Simple Very Deep Convolutional Network for Robust Hand Pose Regression from a Single Depth Image,” *Pattern Recognition Letters*, vol. 119, pp. 205–213, 2019, deep Learning for Pattern Recognition. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167865517303872>
- [14] F. Gomez-Donoso, S. Orts-Escolano, and M. Cazorla, “Accurate and Efficient 3D Hand Pose Regression for Robot Hand Teleoperation Using a Monocular RGB Camera,” *Expert Systems with Applications*, vol. 136, pp. 327–337, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0957417419304634>
- [15] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.
- [16] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [17] X. Chen, G. Wang, H. Guo, and C. Zhang, “Pose Guided Structured Region Ensemble Network for Cascaded Hand Pose Estimation,” *Neurocomputing*, vol. 395, pp. 138–149, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0925231219309087>
- [18] L. Ding, Y. Wang, R. Laganière, D. Huang, and S. Fu, “A CNN Model for Real Time Hand Pose Estimation,” *Journal of Visual Communication and Image Representation*, vol. 79, p. 103200, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1047320321001279>
- [19] L. Ge, H. Liang, J. Yuan, and D. Thalmann, “Robust 3D Hand Pose Estimation in Single Depth Images: From Single-View CNN to Multi-View CNNs,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3593–3601.
- [20] M.-Y. Wu, P.-W. Ting, Y.-H. Tang, E.-T. Chou, and L.-C. Fu, “Hand Pose Estimation in Object-Interaction Based on Deep Learning for Virtual Reality Applications,” *Journal of Visual Communication and Image Representation*, vol. 70, p. 102802, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1047320320300523>
- [21] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” in *European conference on computer vision*. Springer, 2014, pp. 818–833.
- [22] A. Toshev and C. Szegedy, “Deeppose: Human Pose Estimation via Deep Neural Networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1653–1660.
- [23] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

- [24] OpenKinect. (2012) Libfreenect. [Online]. Available: <https://github.com/OpenKinect/libfreenect>
- [25] C. R. Harris and K. J. Millman, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: <https://doi.org/10.1038/s41586-020-2649-2>
- [26] A. Assi, B. Shanwar, and N. Zarka, “detection of abandoned objects in crowded environments,” 06 2016.
- [27] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 3rd ed. Pearson, 2009.
- [28] F. Chollet *et al.* (2015) Keras. [Online]. Available: <https://github.com/fchollet/keras>
- [29] L. dos Santos, “Making Convolution Faster.” [Online]. Available: <https://leonardoaraujosantos.gitbook.io/artificial-intelligence/>
- [30] N. Curti, “NumPyNet Neural Network.” [Online]. Available: <https://github.com/Nico-Curti/NumPyNet>
- [31] M. A. et al., “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,” 2015. [Online]. Available: <https://www.tensorflow.org/>
- [32] P. Dawkins, “Equations Of Planes.” [Online]. Available: <https://tutorial.math.lamar.edu/classes/calciiii/eqnsofplanes.aspx>
- [33] J. de Vries, *Learn OpenGL*. Netherlands: Kendall & Wells, Jun. 2020.
- [34] M. Jones and J. Rehg, “Statistical Color Models With Application To Skin Detection,” in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 1, 1999, pp. 274–280 Vol. 1.

Part 4. Appendix: technical documentation

HARDWARE part of the project

Record 1. System block diagram

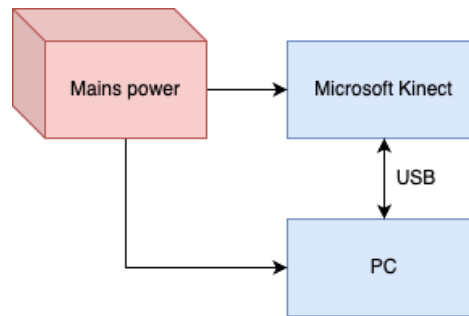


Figure 51.
Hardware system block diagram.

Record 2. Systems level description of the design

The Microsoft Kinect sensor is connected to both mains power and the PC via USB. The PC is connected to mains power too and displays the output of the system and the augmented reality application. The Kinect sensor provides RGB and depth data input to the system and transfers this information in real-time to the PC and the software running on it, where visual and text-based output are shown in a GUI.

Record 3. Complete circuit diagrams and description

Not applicable to this system.

Record 4. Hardware acceptance test procedure

- Plug in the Microsoft Kinect sensor to mains power and the PC via USB.
- Ensure that the green LED on the front of the Kinect is flashing green.

Record 5. User guide

- Plug in the Kinect to power and a PC USB connection.
- Place the Kinect on a stable surface and point its camera at the surface of a flat and empty table at least 1 metre away from the camera.
- Ensure that the camera is positioned so that the calibration rectangle on the calibration screen covers a part of the table surface.
- Move a hand and make the relative gestures at least 40cm in front of the Microsoft Kinect to interact with the virtual object successfully.

SOFTWARE part of the project

Record 6. Software process flow diagrams

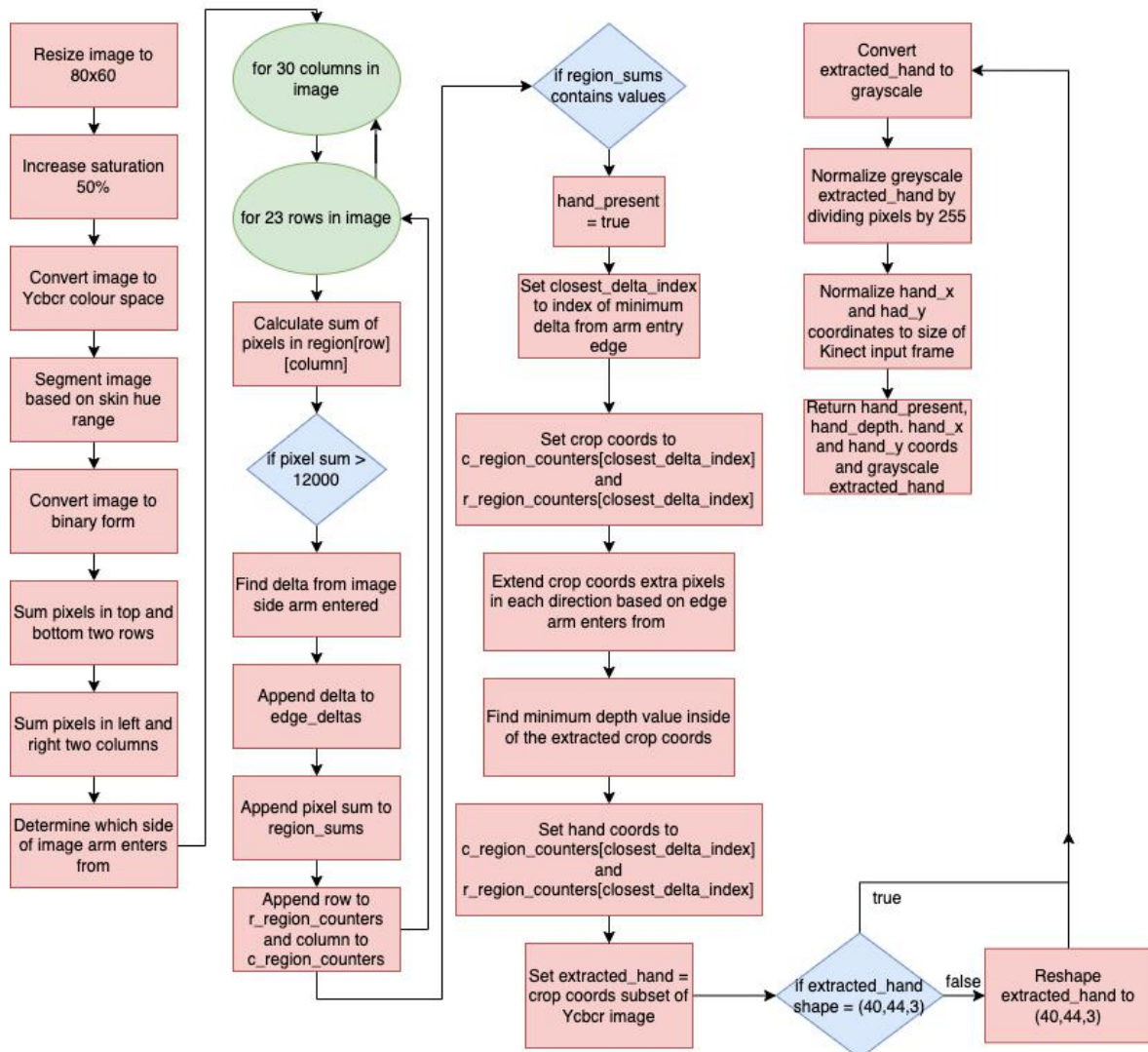


Figure 52.
Flow diagram for the hand detection and tracking algorithm.

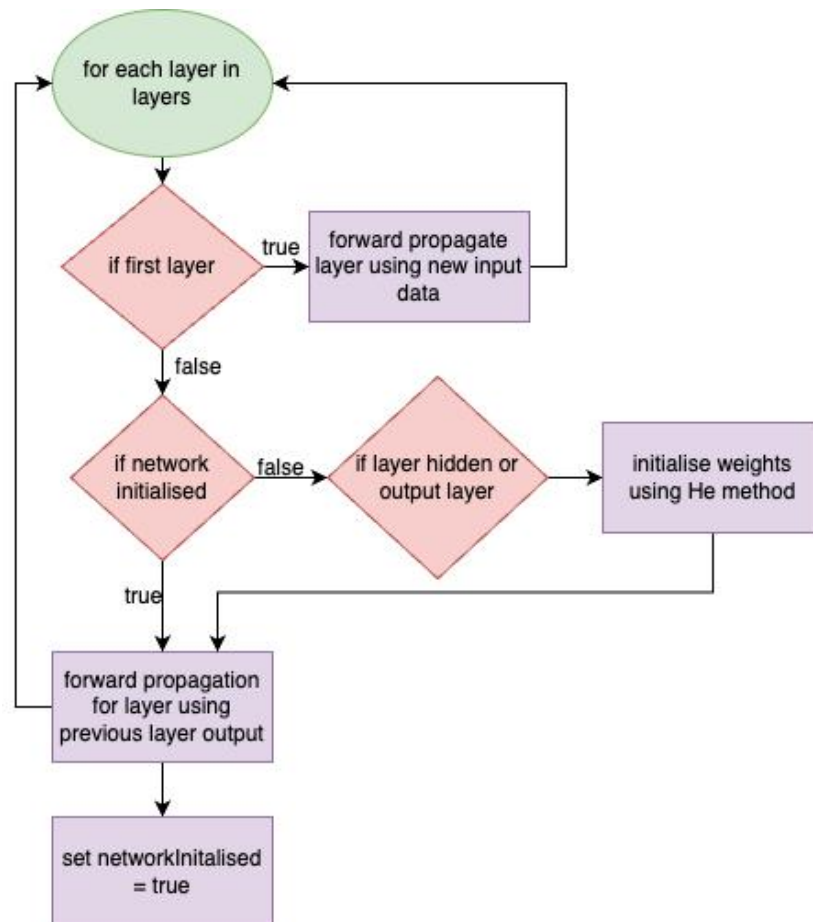


Figure 53.
Flow diagram for forward propagation of network.

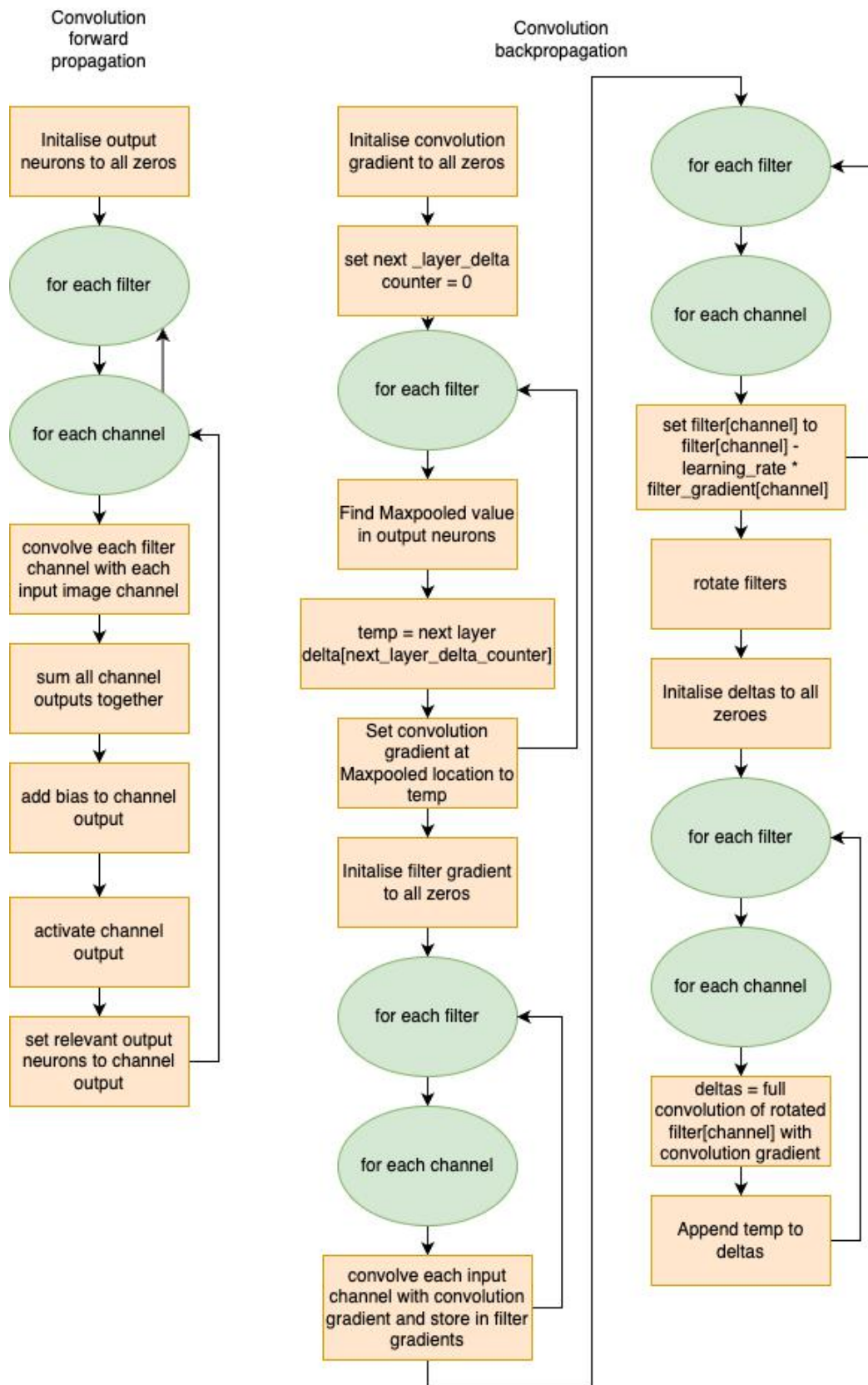


Figure 54.
Flow diagram for the Convolution layer of the network.

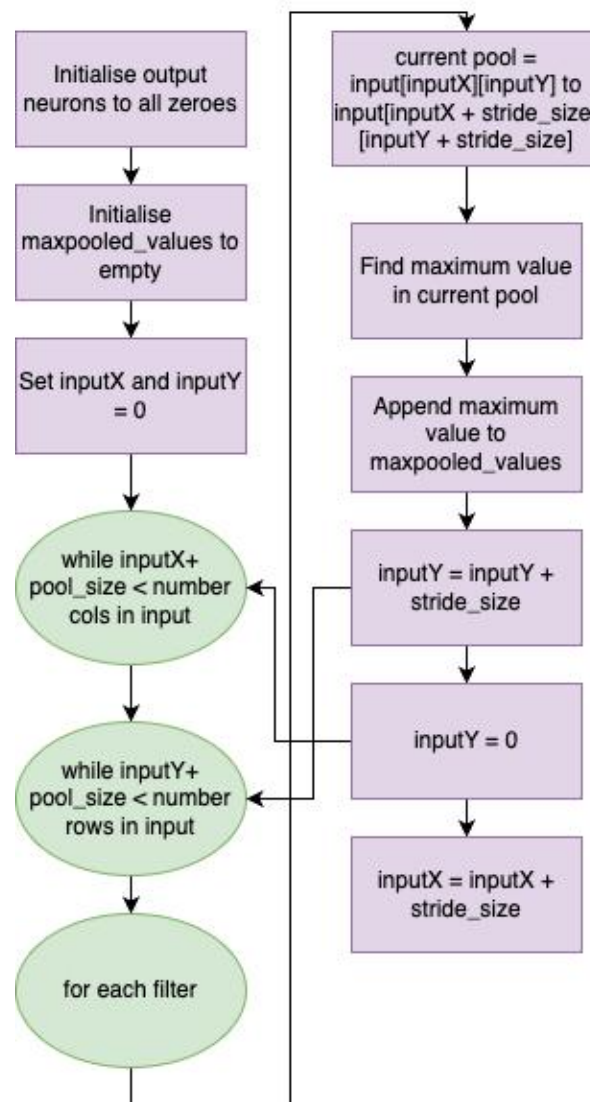


Figure 55.
Flow diagram for the forward propagation of the Maxpooling layer of the network.

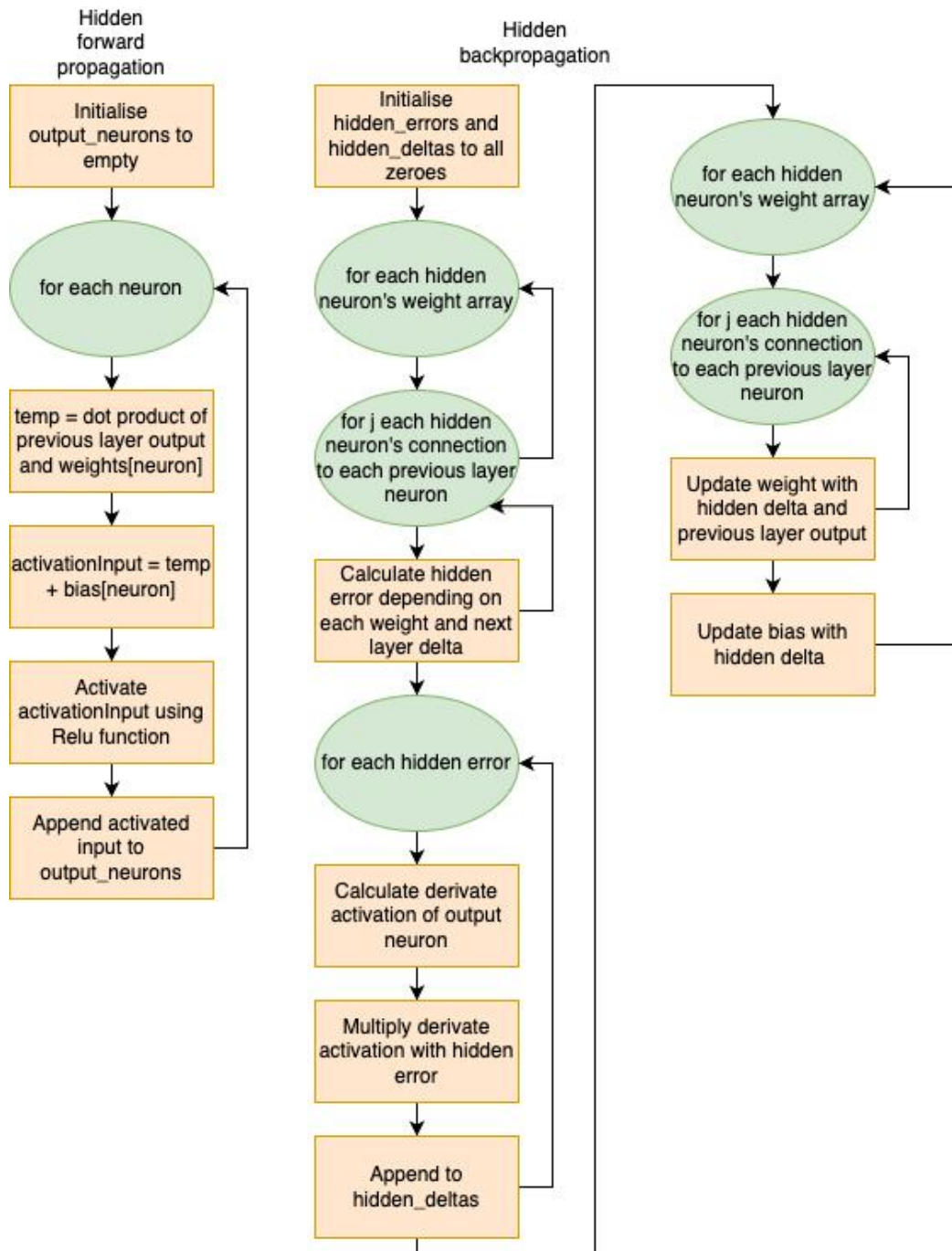


Figure 56.
Flow diagram for the Hidden layer of the network.

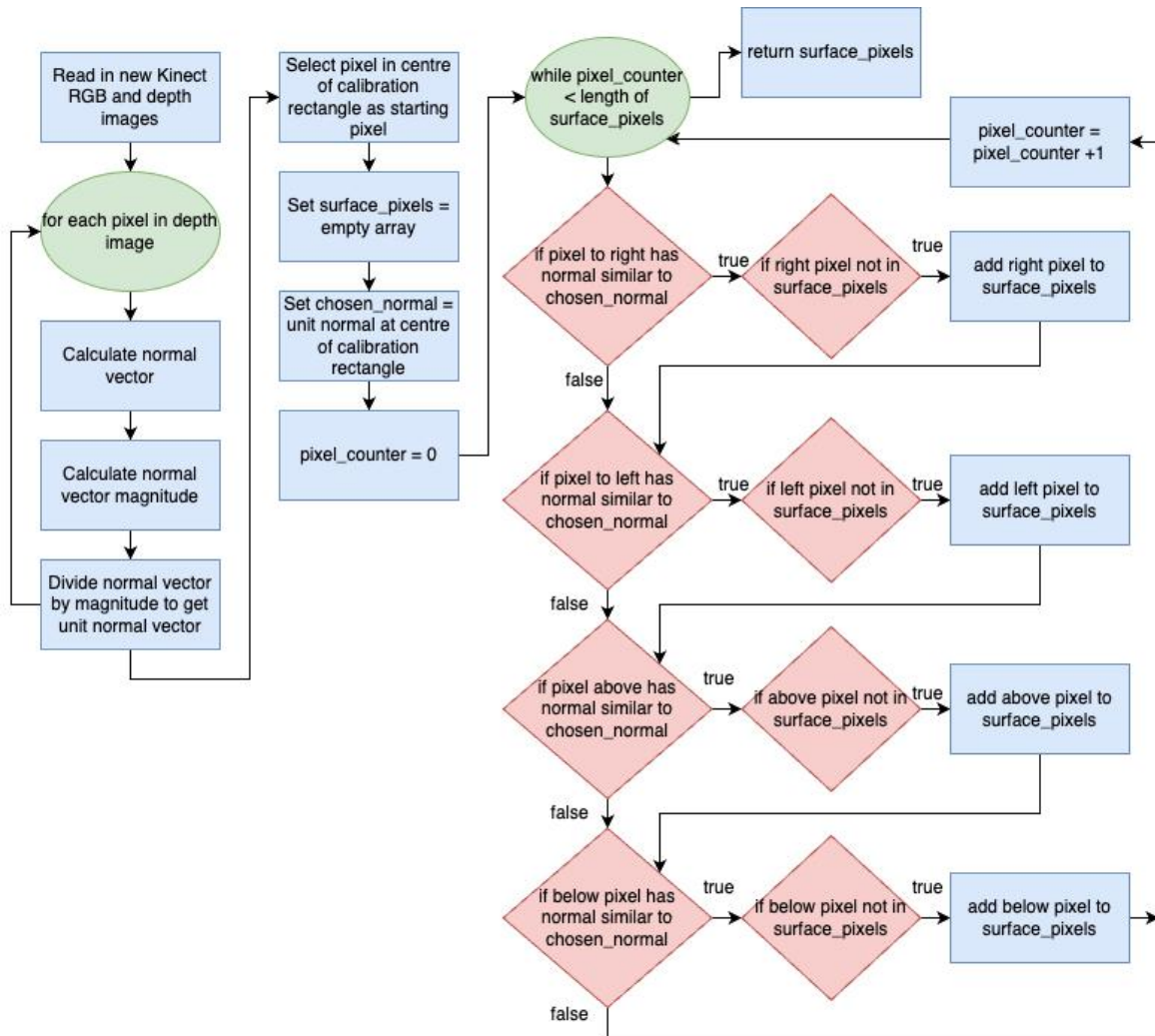


Figure 57.
A flow chart of the surface calibration algorithm.

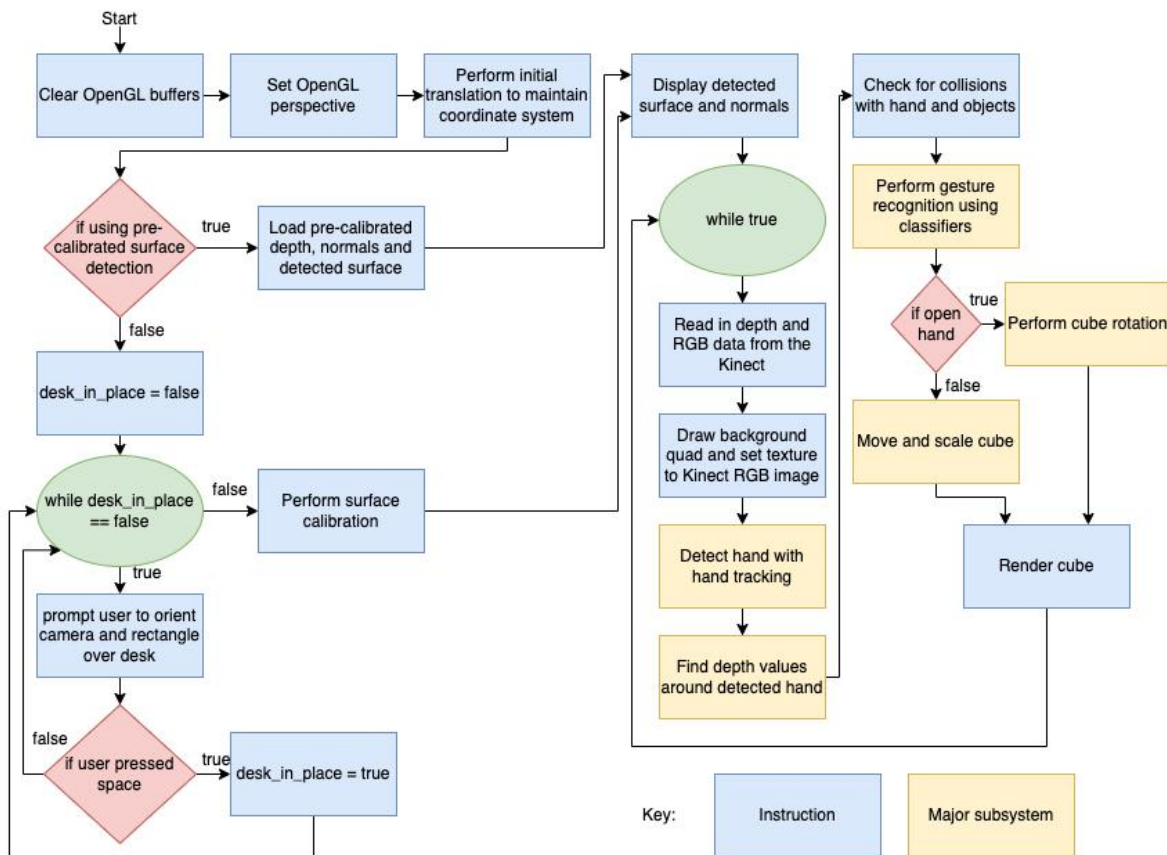


Figure 58.
Flow chart of the main loop operation.

Record 7. Explanation of software modules

The software developed for the system includes:

- A main Python file that runs the augmented reality application, gesture classifiers, virtual object rendering as well as environment recognition algorithms.
- Several Python files that implement the training of the gesture classifiers using datasets of captured images.
- A modified Python file that runs the augmented reality application on the Lattepanda V1 embedded platform.
- Several small data capturing applications for extracting and saving data from the Kinect in an ordered manner as well as augmenting this data with rotations where needed.
- A colourspace range-finding program for finding the correct range of the YCbCr colour space that can segment human skin hue.
- Additional small prototypes used for testing such a C implementation of Maxpooling and comparison program to contrast the performance of the first principles and TensorFlow neural network.

Record 8. Complete source code

Complete code has been submitted separately on the AMS.

Record 9. Software acceptance test procedure

- Run the command **sudo freenect-glfwview** in the terminal.
- Close the pop-up window showing the Kinect input feeds.
- Launch the virtual object AR program.
- When the software is launched, startup messages from Pygame will appear.
- If calibration is enabled, move the camera to the correct position as per the on-screen instructions and then press space.
- Wait while the calibration is performed and observe the console.
- When the calibration is finished, numerous windows will open in the background.
- Open the main AR application GUI to observe the virtual object and interact with it.
- If all the windows are displaying content and changing when a hand is moved in front of the camera, the software is running correctly and the system can begin to be used.

Record 10. Software user guide

- Having connected the Kinect to power and USB, open a terminal on the PC and run the command **sudo freenect-glfwview** to initialise the Kinect sensor and ensure the LIBUSB driver can claim the device. The command may have to be run many times in order to successfully sync the Kinect and the libfreenect control library.
- When the Kinect is successfully connected, a window will appear showing both the RGB and depth camera input from the Kinect sensor. Close this window by entering **Control + C** on the keyboard.
- Modify the program to either use precalibrated surface detection values or acquire new ones by modifying the `use_precalibrated_values` boolean.
- Start the virtual object AR program.
- Take note of the location of the calibration rectangle on the calibration screen (Figure 59), and move the camera so that the rectangle covers a part of the table surface according to the on-screen instructions.
- Move your hand around in front of the camera and make the relevant gestures to control the virtual object on-screen in the GUI (Figure 60).
- Press **Control + C** on the keyboard or close the window displaying the Kinect view and virtual object at any time to quit the program.



Figure 59.
The surface calibration screen of the GUI.

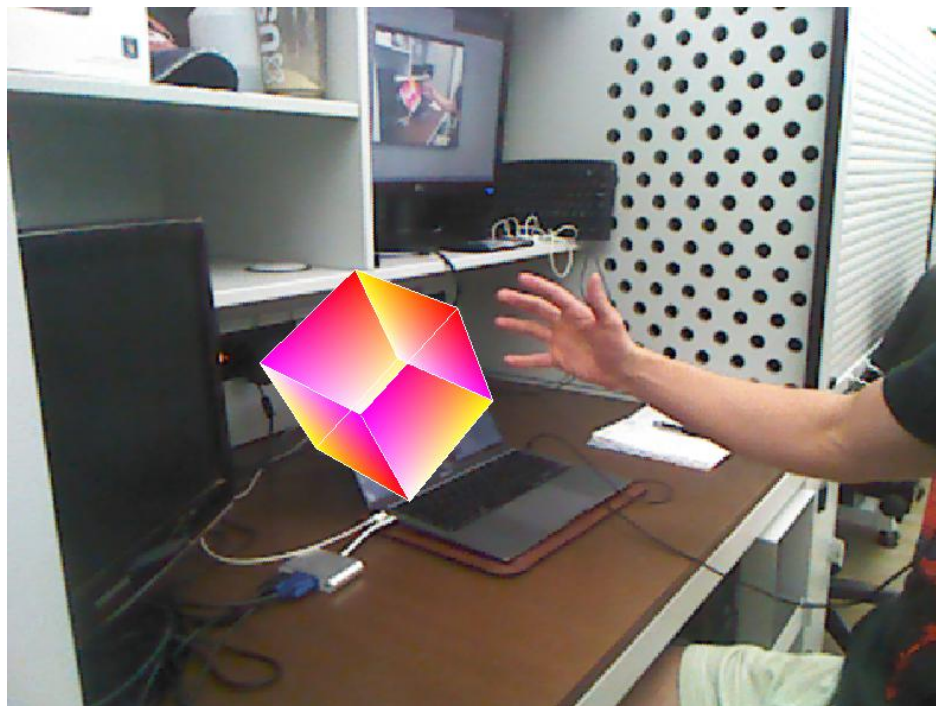


Figure 60.
The main GUI screen showing the user's hand controlling the virtual object.

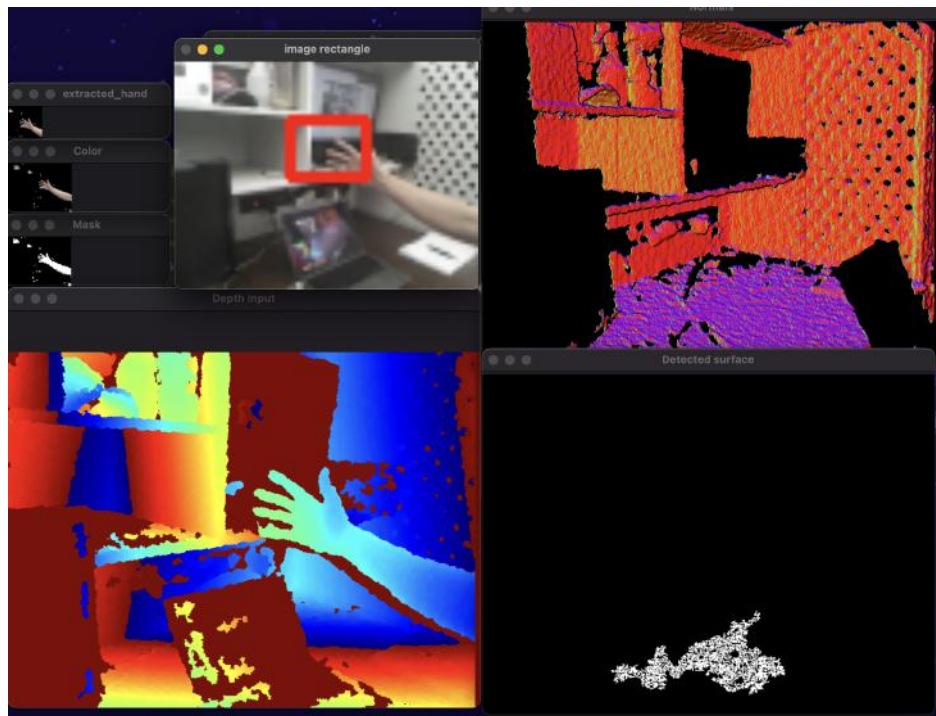


Figure 61.
The additional GUI windows showing information such as normal vectors, depth imagery and hand tracking.

EXPERIMENTAL DATA

Record 11. Experimental data



Figure 62.
A subset of the 1600-image open hand, closed hand dataset.

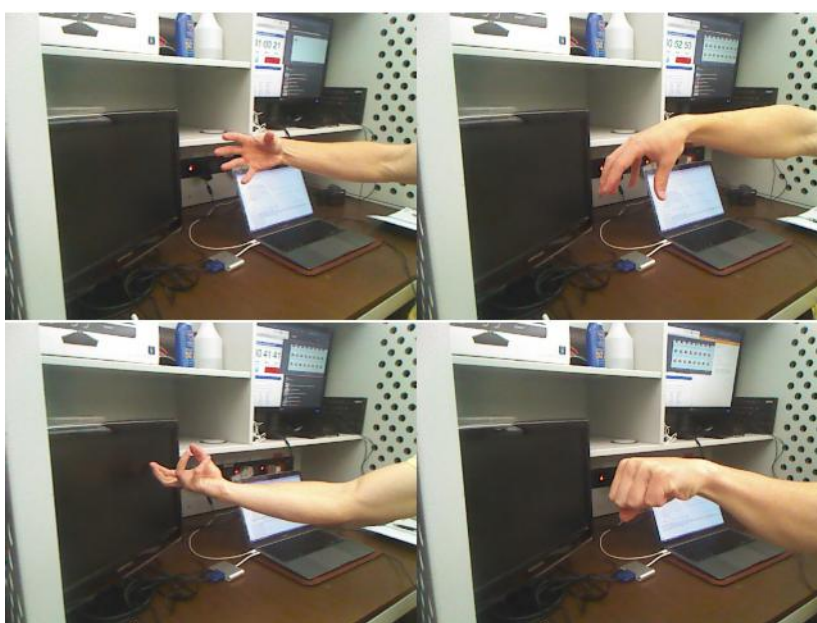


Figure 63.
A subset of the 4200-image downwards, sideways or upwards hand dataset.



Figure 64.
A subset of the 900-image left, towards, right hand dataset.



Figure 65.
A subset of the 1800-image down, side, upwards-facing hand dataset.