UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

FACULTY OF ENGINEERING, BUILT ENVIRONMENT
AND INFORMATION TECHNOLOGY

DEPARTMENT OF ELECTRICAL, ELECTRONIC AND COMPUTER ENGINEERING
http://www.up.ac.za/eece

LAB BOOK

Compiled by

Mitchell Luke Williams
18013555

Generated on

Monday 10th October, 2022

ISG@UP
INTELLIGENT SYSTEMS GROUP
http://isg.up.ac.za/

Contents

1

March 2022

Wednesday, 30 March 2022

Background reading and research

The construction of a system that can both receive input in the form of human gestures and can output a video stream of a virtual object in augmented reality from a separate video input stream has a number of components. In terms of software, these are the human gesture recognition system, virtual object creation system and output video synthesis system. In terms of hardware two webcams or other form of camera should be sufficient for gesture input and environment input.

Background research was conducted on depth measurement and how to conduct it in embedded environments. The use of ultrasonic emitters and receivers seems to be the principal method used in hobbyist and embedded applications. In more serious academic implementations the software approach of computational stereo has been taken that compares the input of two closely-placed cameras and uses triangulation between them and a pixel of an object to estimate depth of objects in the environment. Du2Net uses modern smartphones with multiple cameras and dual-sensor pixels to compute depth. [1] Serious business endeavours use LIDAR sensors and radar to create three-dimensional maps of the environment around them. Certain industry implementations of depth sensing such as Tesla Autopilot rely almost entirely on computer vision using regular cameras and a small host of ultrasonic sensors to construct a representation of the environment around the car. This approach requires massive neural networks that can match training image data to training depth data and thus estimate depth reliably using computer vision only.

Depending on the kind of implementation required it may be necessary to use hardware such an ultrasonic sensor to estimate depth and the surroundings of the environment the virtual object must be placed in, or it may be able to be done entirely using computational computer vision approaches.

There are a few online tutorials and online projects that perform facetracking and object recognition using hobbyist boards like Arduino but it is actually a computer that performs the computer vision and then just sends instructions to a servo motor or actuator connected to the microcontroller. Actual computer vision requires computation that isn't available on small embedded platforms like Arduino boards. There are thus limited options to chose from that suit a budget of R3000 and allow embedded video processing - this is the critical decision, what hardware will allow this system to be constructed such that it performs adequately in real time -

the most important operating characteristic.

Google's ARCore is a software development kit that allows augmented reality applications to be built on many platforms. It contains three main capabilities - motion tracking, environmental understanding of surfaces and light estimation. It conducts many of the high-level tasks needed to implement the system described in the project proposal and is an off-the-shelf solution that should be investigated for inspiration/technical direction.

A point cloud is a set of data points in space that represents an object or environment on a three-dimensional Cartesian grid. It can be used for surface reconstruction which builds a model of a set of surfaces in memory or for terrain reconstruction - basically three-dimensional topographical maps of an environment. Constructing a point cloud of the environment the virtual object is to be placed within would be a helpful first step as the correct scaling and placement of a virtual object could then be computed using those point cloud depth datapoints. Online construction of a three-dimensional model from point maps is difficult as this requires a lot of computation and several industry implementations still use offline training - just because it is cheaper and more accurate. Google Cartographer uses online training but requires at least a high-end Intel i7 chip to run the required algorithms. Potentially if the resolution and quality of the model is reduced it could be done on an embedded platform and there are examples of this such as Angry Birds AR on smartphones.

Thursday, 31 March 2022

Prototype ideas

A list of prototype task ideas that could aid in deciding on further specifications for the system to built are as follows:

- A small image recognition system that could identify at least 1 distinct hand gesture on a small embedded platform like an ESP32
- Depth point cloud creation of 10 points using a microcontroller like an Arduino with a hobbyist ultrasonic sensor
- Generation of a shape onto a JPEG on a PC

These tasks would aid in deciding on further specifications because it would be able to estimate how much processing power is needed to run a computer vision application that can correctly interpret hand gestures, how much memory it takes to store a depth point cloud as well if a physical rotating device is needed to move the ultrasonic sensor around to accurately map a scene - like a LIDAR sensor. These tasks might even be impossible as the processing power of these hobbyist electronic devices are really limited (ESP32 has a max of 240MHz clock) and so even attempting them will inform the design decisions to be elucidated in the project proposal. Hobbyist boards are suggested here simply because they are available and affordable and do not impact the project's budget. It is also uncertain if a more expensive LIDAR or Microsoft Kinect-like sensor is needed to accurately interpret the depth of an environment or if computer vision-only approaches will yield enough information for the system to operate accurately. Finally it is also imperative to see the graphical computations involved in drawing a shape over a still image and predicting how that would scale to processing images say at 30 frames per second required to insert a virtual object into a real live video feed.

Literature Analysis

The process of identifying hand gestures has been well studied and generally consists of a number of steps. In the hand gesture recognition system built by Shen et al. [2] the process includes segmenting the region of the image that just contains a hand from the background imagery, finding and then keeping track of certain components of the hand such as fingertips and the deformations between the bottom of the fingers (convexity defect points) as well as then determining the relative position and orientation of the detected hand to the camera capturing the image to estimate where and at what scale an object to be virtually created should be rendered.

Hand segmentation in [2] is conducted using nonparametric skin color modelling in order to perform well for multiple different skin colors and in varying light conditions. The Continously Adaptive Mean Shift algorithm is used for this by recognizing very high or very low saturation of image pixels and since human skin hues are all very similar will be able to pick them out from background imagery. The same study uses a curvature-based algorithm for detecting features of the hand such as fingertips. This encompasses calculating vectors on the hand pixels based on the hand contours and a least-square fitting method. If the system running the hand segmentation and detection process is fast enough the entire process can be repeated for each frame of video to be analyzed but if it is not, tracking and predicting where in the next frame the points of the hand will be is required. Shen et al. utilize a "P4 3 GHz PC equipped with 1 GB RAM" to run their hand gesture recognition algorithm.

A slightly different approach is taken by University Of Canterbury's Human Interface Technology Lab hand gesture control system [3] where a Microsoft Kinect is used to reconstruct the surface of a tabletop using point cloud information from the Kinect and allows virtual objects to interact with human hands detected by skin-colored regions of the image. The resulting virtual object is displayed either on an Android mobile phone

or desktop PC.

Other approaches include utilizing depth-sensing hardware such as a Leap Motion sensor to detect depth and construct a hand skeletal model [4] but this implementation is off-the-shelf in that the hand tracking is performed by the Leap motion itself and outputs a live model of the user's hands from the sensor. In Lee et al.'s implementation of gesture control of an object [5] a standard camera is used to predict a six-degree-of-freedom camera pose also with a similar skin-color segmentation algorithm that can identify fingertips and thus build a virtual model of the hand. Their implementation includes optical flow tracking calculations and multithreading for predicting future hand positions based on current ones - a useful solution to a noisy environment or under-powered system.

Several papers allude to the seminal "Statistical Color Models with Application to Skin Detection" paper [6] that outlines how different human skin color is very different in intensity or saturation but not hue. Hue is a pure color and intensity is the saturation/brightness of a colour. Thus skin and non-skin colors can be mapped into a histogram model of skin and non-skin pixel colors which can then be applied in the segmentation step to differentiate skin versus non-skin pixels in an image. It is undetermined if the data for a skin classifier must be gathered from first principles or if it can be used off-the-shelf.

The use of hand gestures to control other devices has even been extended to the control of drones [7]. This is notable because it operates in real-time and on an embedded albeit expensive platform - the Microsoft Hololens. This system uses four RGB cameras as well as a time-of-flight sensor to track and interpret the user's hand gestures. The Hololens contains a Qualcomm Snapdragon 850 mobile CPU and points to the similar conclusion as the other papers discussed above that the computation of augmented reality applications relies on still computationally heavy processors - not cheap embedded systems. The drone system performs gesture control by analyzing the hand joint's position, angles between the fingers and "co-linearity between finger pairs" to determine the current state of the operator's hand and interprets these as commands for the drone. The system ran video processing on the Hololens device and with custom FFmpeg decoding tools to process frames of video.

Researchers at Google have used MediaPipe and in-house developed algorithms to accurately track the hand and gestures of a user using just an RGB camera. [8] Their two main algorithms include a palm detector which segments the hand from the rest of the image and a hand landmark model which predicts the shape of the hand skeleton based on the segmented image and predicts the position of 21 hand-knuckle points. Their solution trains the hand skeleton classifier on a dataset of thousands of images of hands in various gestures with the skeleton model annotated on the image. This is a more deep-learning approach than detailed segmentation and finger convexity calculations like in the Shen et al. study [2]. The Google researchers produce smaller and more lightweight versions of their model for use on smartphones and web applications as well.

Impact on system to be developed

The literature studied at this point points to a number of conclusions for the system to be developed. Firstly, the hand gesture recognition algorithm will require building some kind of model of a hand if anything more than a few discrete gestures are to be identified by the system and if continuous control of an object is to be implemented. A hand skeleton model or some other Cartesian coordinate system needs to be mapped onto a detected hand and live-updated to represent gesture input. Secondly, in order to build a system that intelligently recognizes the scale and depth of an environment presented to the system via video input some form of depth measurement will need to be taken of the environment. Certain applications operate on vision-only input but

these are computationally expensive and not as accurate or reliable on embedded hardware as applications that combine regular RGB camera input with either time-of-flight or ultrasonic depth data of an environment. Thirdly, the computation of all of these algorithms will be extremely heavy if the system is to operate in real time and thus either the system needs to perform its computation on a PC with respectable specifications for graphics and matrix mathematics or the subcomponents of the system need to operate on different embedded platforms - one device for the hand gesture recognition algorithms and one device for the environment detection, graphical creation and combination algorithms.

2

April 2022

Saturday, 02 April 2022

Initial prototypes

A small prototype was built using Teachable Machine to see if a small neural network could serve as the basis for the hand gesture recognition system if discrete gestures were used. Most of the research papers reviewed so far generate a three-dimensional model of the hand which is likely necessary for fine-grained control of the virtual object and will need to be implemented. But it is also uncertain if the segmentation and explicit hand position calculation approach is going to be necessary given how successful Zhang et.al's deep-learning only approach [8] to generating a three-dimensional hand model is.

Fig. **[2.1]** shows an open-hand input to the teachable Machine prototype and Fig. **[2.2]** shows the closed-hand input. The two different hand gestures either rotate a cube to it's leftmost face or topmost face. The algorithm works in a web browser and using a webcam for input can accurately rotate the cube based on hand input. Thus, it is relatively trivial to implement discrete gesture recognition using off-the-shelf software like Tensorflow with basic neural networks trained on small amounts of data (less than 1000 images). The prototype is extremely simple and because it is not optimized or trained with lots of data it breaks down when seeing new input - like an open hand but with the user wearing a mask but demonstrates that small neural networks can be used for discrete gesture recognition.

The rendering of a three-dimensional cube or similar object will require drawing over each frame of video. Representing each pixel of an image in a three-dimensional array of width x height x pixel values will allow each pixel to be changed as needed when a virtual object is "inserted" into the video. Using Python and the Pillow image library to convert an image into this array format, a simple prototype was built to render a three-dimensional cube onto a picture of a room. Fig. **[2.3]** shows the output of this algorithm which just draws a small green cube over a random area of the image. Scaling this up to redrawing the cube for each frame of video and a different size and perspective based on the movement of the camera from the initial placement of the object seems feasible.

The use of graphical libraries like OpenGL or engines like the Unity Engine seems to be the industry standard for generating computer graphics and considering this is a critical element of the system it will probably need to be done partly off-the-shelf in order to generate realistic virtual objects. Rendering the virtual object as was

Figure 2.1: Open-hand input to TeachableMachine gesture recognition prototype

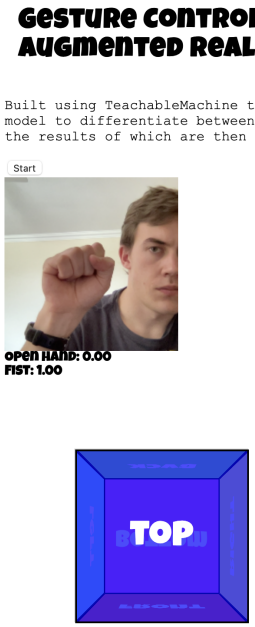


Figure 2.2: Closed-hand input to TeachableMachine gesture recognition prototype

done manually by replacing pixels in the cube rendering prototype seems inefficient and is predicted to not scale up well to the 24fps rendering needed for the final system. The creation of shadows on the environment is usually the next step in creating realistic lifelike objects but prototyping this needs to wait for depth information to be considered before it can be mapped onto the environment.

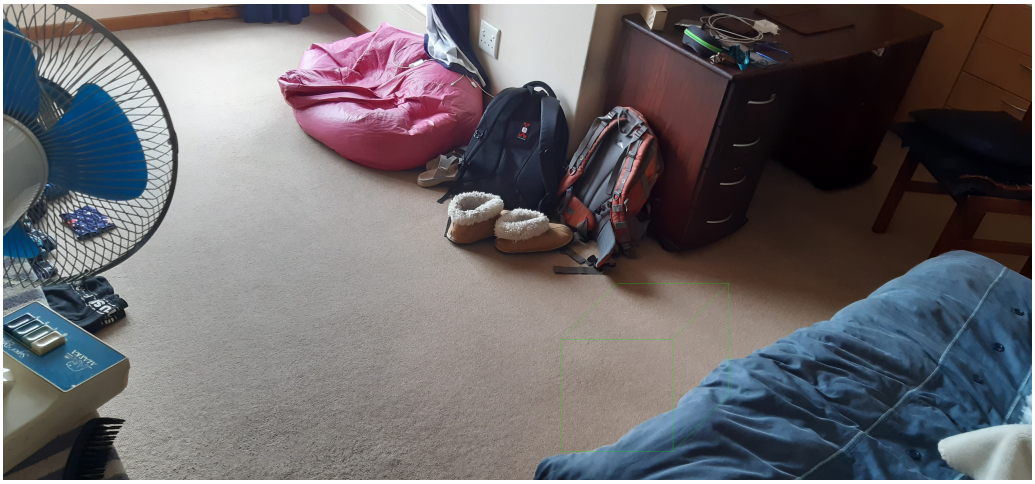


Figure 2.3: Cube rendering prototype output

Sunday, 03 April 2022

Functional block diagram updates

Time was spent refactoring the functional block diagram for the project proposal, splitting the system into four main processing components - the gesture recognition algorithm, the virtual object control algorithm, the environment recognition algorithm and the augmented reality creation. The second draft of the diagram can be seen in Fig. [2.4]. More detailed breakdowns are possible but this presents a high-level overview of the system and takes the approach of building a detailed skeletal model of the user's hand and a detailed depth map of the environment in which to place the virtual object. The Virtual Object Control subsystem was implemented as a template-matching system due to the constraints of memory and the advantage of being able to precompute and store certain gestures as templates for later use and identification.

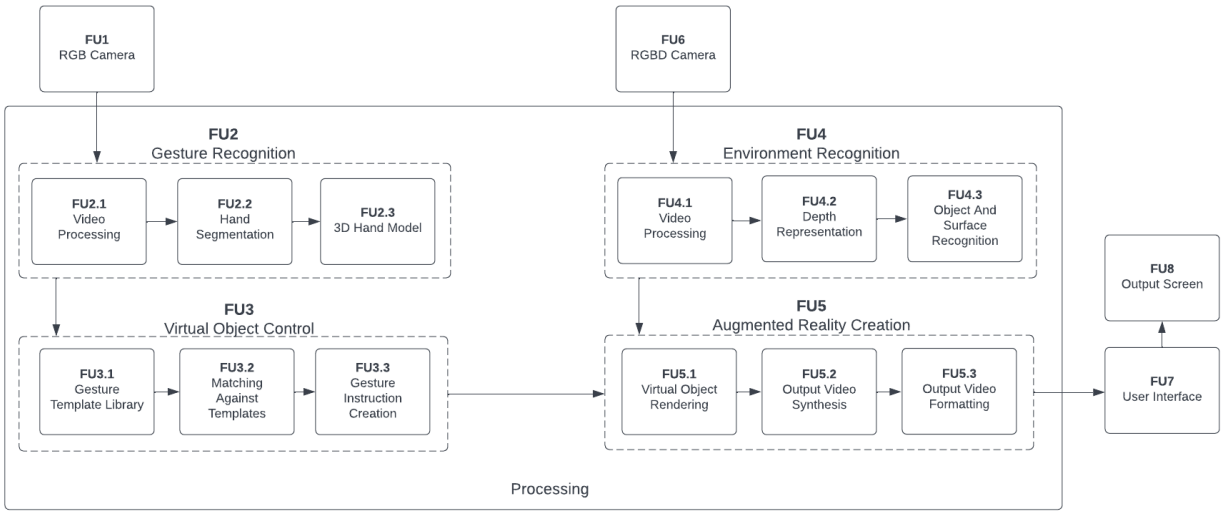


Figure 2.4: FBD draft 2

Initial hardware prototypes

In order to gain familiarity with ultrasonic sensors and depth measurement systems that might be needed to accurately map the environment around the system, a prototype was built using an HC-SR04 ultrasonic distance sensor with an Arduino Uno. The sensor was pointed around a roughly 4x4 metre room and different measurements taken to see if a small sensor like this could be used to generate a depth point cloud of an environment. The HC-SR04 ultrasonic sensor is a hobbyist sensor that has a maximum range of 4m and sends out a 40kHz sound wave out of an ultrasound transmitter and then receives any reflected sound waves on its ultrasound receiver. The time difference between sending out a sound wave and receiving one back can be used with the speed of sound in air (343m/s) to calculate the distance between the sensor and an object. Fig. [2.5] shows the ultrasonic sensor connected to the Arduino Uno and Fig. [2.6] shows how a whiteboard was used to reflect the transmitted sound waves back at the sensor from varying distances. These distances were steadily increased from an initial 4cm as the whiteboard was moved further away from the sensor and a graph of these results is provided in Fig. [2.7].

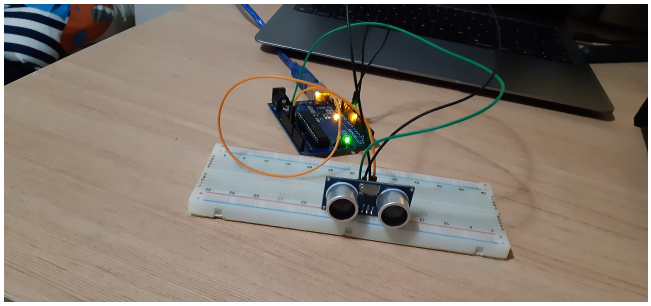


Figure 2.5: The ultrasonic sensor connected to an Arduino Uno

Figure 2.6: The ultrasonic sensor measuring test

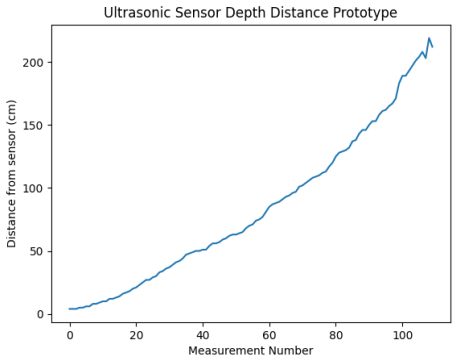


Figure 2.7: Graph of measuring experiment using HC-SR04

The measurements became too noisy to be useful past 2m but the sensor performed well before this. However, it is noted that the sensor did not reliably pick up reflected sound waves reflected by a person or other soft objects like pillows and this is theorized to be due to the shallow effectual angle of less than 15deg for the receiver - irregular shapes like human body parts did not reflect the sound wave straight back at the receiver and thus didn't create any input. This is a problem for a virtual object system that must be able to visualize the surrounding environment accurately so that objects can be placed in it and interact correctly with hard and soft objects alike. Similarly, the ultrasonic sensor only projects sound waves in a straight line and thus has a narrow field of vision that limits the depth-sensing range of the device to an almost flat plane in space. This is not adequate for fully mapping an environment around the system and thus a more robust depth-mapping solution like a spinning platform with an ultrasonic sensor on it, LIDAR sensor or Microsoft Kinect's light-emitting depth measurements

needs to be considered.

Monday, 04 April 2022

Project proposal improvements

Time was spent refactoring the project proposal according to Mr. Grobler's comments and completing additional sections. The skeletal hand model was decided upon as the method for determining hand gestures because it has the most fine-grained information for different gestures and can be implemented using an RGB camera only as opposed to requiring depth information. The skeletal model developed by Zhang et al. [8] serves as inspiration as to what can be accomplished using off-the-shelf components such as Tensorflow. The system developed by Zhang et al. was optimized for mobile use using TensorflowLite but still performed best in a desktop application runtime. Thus, this behaviour in addition to numerous other papers implementing gesture control on PCs only leads to the conclusion that the system should be implemented on a PC and not an embedded platform. This is in addition to the graphical demands of creating a virtual object and handling the input of two separate cameras - a task that demands either a very expensive embedded platform that is beyond the budget of this system or a PC.

The system requirements were developed for the project proposal based on the research presented above in this lab book. Additionally, RedShark Cinematography explains that 24fps is the standard frame rate for movies because it is slow enough for the human eye to fill in missing frames and create a moving picture in the brain but fast enough to not appear janky or disjointed - the way silent films of the 1920s appeared due to hand-cranked film cameras. Thus 24fps is the minimum frame rate decided upon for updating of the virtual object and hand model.

The field specifications were similarly designed in addition to the University of Warwick's Lighting Guidance For Offices which states that standard office lighting for computer work should be 300lux. The 2m distance of the virtual object to the RGBD camera comes from the ultrasonic test prototype in which sensor output became too noisy after 2m of distance between the sensor and reflecting object and dimensions of most offices.

Tuesday, 05 April 2022

Embedded neural network prototype

The final prototype to be built from the list written on the 31st of March was "A small image recognition system that could identify at least 1 distinct hand gesture on a small embedded platform like an ESP32." This was implemented by using an off-the-shelf ESP32 Cam example Arduino file that downloads and runs inference on a TensorflowLite Model. The model created previously using TeachableMachine in the Hand Gesture Web demo was used again and loaded onto the device. Fig. **[2.8]** and Fig. **[2.9]** show the results of the model when an open palm and fist are displayed to the camera of the ESP32. The model does not accurately differentiate between the two different gestures shown to it and this is theorized to be because of minimal training data that was all webcam input and now this low-resolution camera input is not matching to any of the expected inputs to the neural network.

Additionally, the model is incredibly slow and can interpret the hand gestures nowhere near 24 times a second - it is estimated generously that it is making a prediction once every second. The ESP32 cam also does not record video at a high-enough resolution to capture hand features well enough that a 21-point skeleton could be modelled. Thus the results of this prototype point towards a much higher resolution camera being needed to provide input for the gesture recognition system as well as more processing power and better training for the computer vision algorithm that is ultimately implemented for the hand gesture recognition.

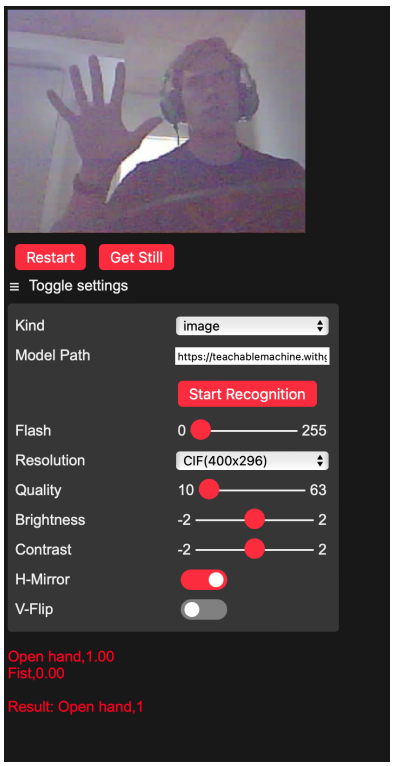


Figure 2.8: Output of the ESP32 web server running the hand gesture model

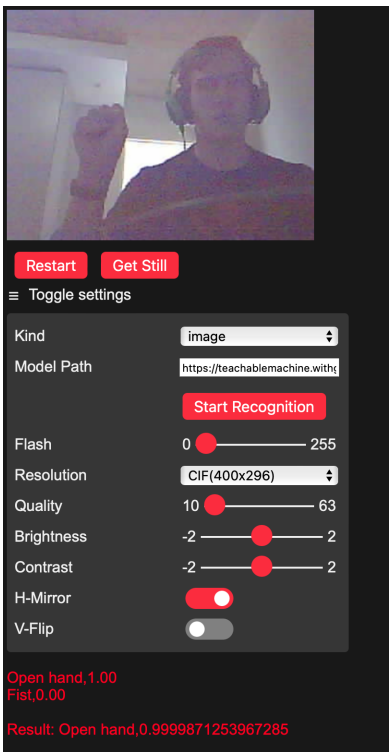


Figure 2.9: Output of the ESP32 web server running the hand gesture model

Thursday, 14 April 2022

Libfreenect demonstration program

Some changes were made to the project proposal today in order to converge upon a third draft of the proposal. However, the salient discovery of the day was discovering the Libfreenect open source library that makes controlling a Microsoft Kinect possible on both Windows, Mac and Linux. Two Microsoft Kinect devices were signed out from Mr. Grobler's hardware store last week and this library appears to contain the drivers necessary to interface with the Kinect on modern PC platforms.

Testing with the library needs to be conducted but it appears wrappers for the API in Python, C, Java and C++ are all available. The choice of programming language will obviously be decided by engineering factors such as the required speed of the system and graphical library support. However, it is predicted that if real time speed is becoming difficult to achieve that C might be needed for its low-level high-performance nature but if Python can provide adequate performance it may be much more utilitarian and conducive to constructing the more complicated parts of the project such as matrix mathematics and 3D scene reconstruction with point clouds. This informs the direction in which to take the design of following prototypes.

Friday, 15 April 2022

Libfreenect testing

The libfreenect library was installed and the demo program that comes with the installation was used to pull basic information from one of the Microsoft Kinects - a RGB image was displayed from the camera and a false color image with different colors representing different depth measurements from the depth camera. A screenshot of this demo running on an Apple Macbook Air M1 is displayed in Fig. [2.10]. However although the demonstration program runs adequately on this machine, installing the library in the Python environment of the laptop proved difficult and ran into many compiler errors since libfreenect is a C library re-optimized for use with Python and the environment of the laptop proved difficult to set up. This installation troubleshooting is still ongoing.

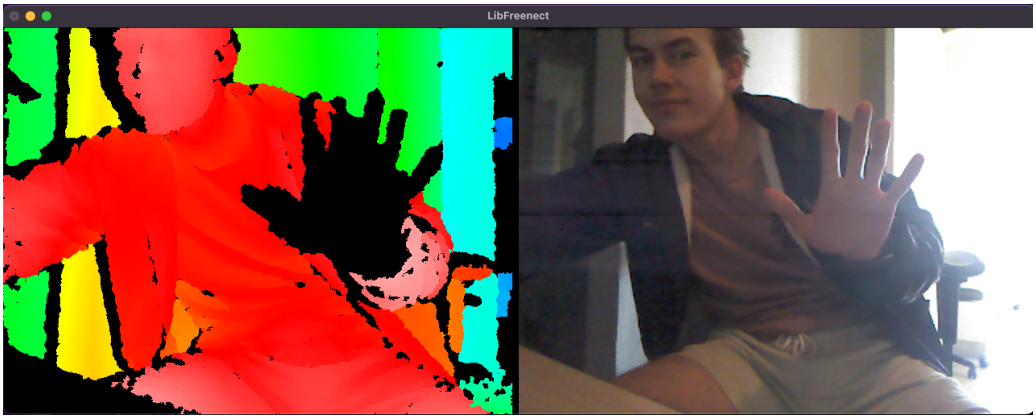


Figure 2.10: Output of the libfreenect demo program

Wednesday, 20 April 2022

Initial Version Of Experimental Plan

Qualification Test 1: Observation Of Result Of Manipulation Of Virtual Object Using Single Hand Gestures

Objective: To confirm the user can manipulate a virtual object using the discrete gestures of a single hand.

Equipment Used: The two input cameras, a ruler and a PC are used.

Experimental Parameters and setup: The two cameras are connected to the PC and the system initialized. The front-facing camera is pointed directly at the user and the rear-facing camera is pointed at the environment directly in front of the user and PC with a big enough flat surface visible so that the virtual object is instantiated onto it. A list of the expected behaviours of the virtual object for the various gesture inputs is generated for comparison against.

Experimental Protocol: The user will hold up their hand and display each discrete hand gesture for controlling the virtual object for two seconds each. These gestures are those for the up, down, left, right, rotate on the x-axis, rotate on the y-axis, rotate on the z-axis, backwards and forwards commands for the virtual object. The observed change in the virtual object will be noted, and its distance moved on the flat surface measured with a ruler. These measurements will be noted against the expected behaviour for these gesture commands.

Qualification Test 2: System Recognition Confirmation Of Hand Gestures In Real Time

Objective: To ensure that the individual hand gestures can be recognized by the system in the time required to ensure that the system operates in user-apparent real-time and that the known gestures of the system can be accessed in the same required time period.

Equipment Used: The two input cameras, a PC and a software timer will be used.

Experimental Parameters and setup: The system hardware will setup and initialized as for Qualification Test 1 and then a software timer added to the code to measure the time between a new frame of video being received by the system to the time a gesture prediction is returned by the system.

Experimental Protocol: The user displays the 9 different gestures for various commands to the virtual object. As this is done the system logs how long each prediction of which gesture is being displayed takes to generate and writes these values to a log file. This generation process checks the current hand gesture against known gestures in memory and returns the predicted gesture. Ensuring that the correct gesture is recognized for each, the software timer logs are examined, and each comparison time ensured to be less than 41.6ms so that the system is recognizing the user's hand gestures at a user-apparent real-time of 24fps.

Qualification Test 3: Testing The System's Physically Accurate Rendering Of A Virtual Object

Objective: To ensure that the system can accurately model a three-dimensional object in augmented reality.

Equipment Used: The two input cameras, a PC, a screenshot tool and a physical 20cm by 20cm cube will be used.

Experimental Parameters and setup: The cameras are setup and the system initialized as in Qualification Test 1. A physical 20cm by 20cm cube is placed onto the flat surface in front of the rear-facing camera.

Experimental Protocol: The physical cube will be moved ten centimeters in each direction and rotated 180 degrees in each planar direction. After each movement a screenshot of the rear-facing input video feed will be taken to acquire a reference image of the physical cube. Then, the gesture instructions for each virtual object

control command will be sent to the virtual object control algorithm manually and a screenshot will be taken of the output video feed of the virtual object in the environment. The two sets of screenshots will be compared to ensure that the virtual cube is accurately modelled in augmented reality in reference to the real-world cube.

Qualification Test 4: Frame Rate Measurement And Real-Time Operation Of The System

Objective: To ensure that the system is updated in real-time and displays output video that appears to have no visible latency to the user.

Equipment Used: The two input cameras and the first-principles frame counter will be used.

Experimental Parameters and setup: The input cameras are setup as for Qualification Test 1 and the frame counter displayed on the user interface.

Experimental Protocol: The user will display the 9 hand gestures for the various commands to the virtual object for 2 seconds each and the frame counters for the virtual object rendering algorithm as well as the hand model creation algorithm will be observed - if they stay above 24fps the system is operating with a latency less than 41.6ms and will appear to run instantaneously to the user in real-time.

Qualification Test 6: Confirming Physical Realism Of Virtual Object Behaviour

Objective: To ensure that the virtual object behaves in the physically realistic manner in which a real object would behave.

Equipment Used: The two input cameras, a PC, a timer, a ruler and a physical 20cm by 20cm cube will be used.

Experimental Parameters and setup: The two input cameras are setup as for Qualification Test 1 and the physical cube will be placed on a flat surface in front of the rear-facing camera.

Experimental Protocol: The system will be initialized so that the virtual cube object appears on the flat surface and no input commands will be provided to the system. The virtual cube should remain stationary on the flat surface for 10 seconds before the user attempts to move (using gesture control) the object past the right of the physical cube first with 30cm of distance between the virtual cube and the physical cube, then 20cm, 10cm and 0cm. This process is then repeated for the left, right, backwards and forwards commands with the physical cube being moved to the position where it stands in the path of the virtual object. The virtual object's position should be noted at each attempted movement and its movement observed - the virtual object will only be able to move past the physical object when 10cm or more space exists between them, measured by the ruler.

Saturday, 30 April 2022

Image thresholding prototypes

The aim of the work session today is to experiment with thresholding, image segmentation and image processing to estimate what kind of methods will be needed for the final system to function and effectively derive information from a webcam and convert that into gesture input. To that aim, the OpenCV and PIL libraries were installed onto a PC and a Python program implemented. This program converted a webcam image into an array of pixel values and then scanned through all the columns and rows of the array pixel by pixel and evaluated if the three RGB pixels were close to the RGB values of the colour red. If they were, the algorithm left them be else the pixel was set to white and the algoirthm proceeded to the next pixels.

In this way, a red thresholding algorithm was implemented and attempted to pull out just the parts of the webcam image that were red. Red was chosen in this case because it is a striking color and none of the background items at the time contained much red in them, while the foreground item - a person wearing a red hoodie did. The results of the algorithm are presented below in Fig. [2.11] and Fig. [2.12]. It is clear that the shadows and shades of the red hoodie make it difficult to get an accurate thresholded shape and comparing pixels on an individual basis misses a lot of what is actually trying to be thresholded. Considering pixels and the average of a few pixels is predicted to be a better method of doing this with a kernel-based approach. But even so, the color-based thresholding is not suitable for the hand segmentation approach as people's hands are all different colors and different parts of the body have the same color. Thresholding with a grayscale image and specific hand segmentation approaches will next be investigated. Greyscale images are where the value of a pixel merely represents the amount of light present in that pixel and is often used to separate background and foreground information because background pixels are usually less intense than foreground images which are closer to a camera

Figure 2.11: Webcam input to the red thresholder algorithm

A greyscale thresholding implementation is now tested. The image is converted to grayscale and five different threshold algorithms used - these are the binary, inverted binary, truncated, truncated to zero and truncated to zero inverted thresholding functions. The results of these implementations are visible in Fig. [2.13] to Fig. [2.17]. The inverse binary threshold algorithm performs the best in terms of delineating a human form from background imagerey. These tests were performed in a relatively dark room and environmental factors are to be considered when implementing a thresholder that must function in various lighting conditions.

Figure 2.12: Output of the red thresholder algorithm

Figure 2.13: Binary threshold algorithm

Figure 2.14: Inverse Binary threshold algorithm

Figure 2.15: Trucated threshold algorithm

Figure 2.16: Truncated Zero threshold algorithm

Figure 2.17: Inverse Truncated Zero threshold algorithm

3

May 2022

Friday, 06 May 2022

Libfreenect library installation

Multiple hours were spent attempting to install the necessary libraries to enable communication with the Microsoft Kinect camera and the Macbook Air M1. The principle problem appeared to be installing the C-compiled libfreenect library into a useable Python module that could be installed into the virtual "kinect-project" anaconda environment being used for the prototyping. The libfreenect library first had to be compiled and built locally - which was achieved after modifying a few settings but then had to be installed by running a setup.py file to incorporate these C files into a Python wrapper module that could be used from other Python code. This installation failed nearly one hundred times as it could not find the requisite C libraries on the filesystem. After much trial and error, the installation eventually succeeded and the libfreenect library was made available for interfacing with a Kinect connected to the laptop. The successful import of the libfreenect library in a Python file on the laptop is visible in Fig. **[3.1]**.

The steps to connect the Kinect to the laptop and successfully interact with it in Python code are as follows.

- Plug in Kinect to power and USB
- Run in Kinect-project conda environment
- If can't claim USB error occurs - run sudo freenect-glview first
- Run kinectBasics.py or other file with libfreenect commands in it

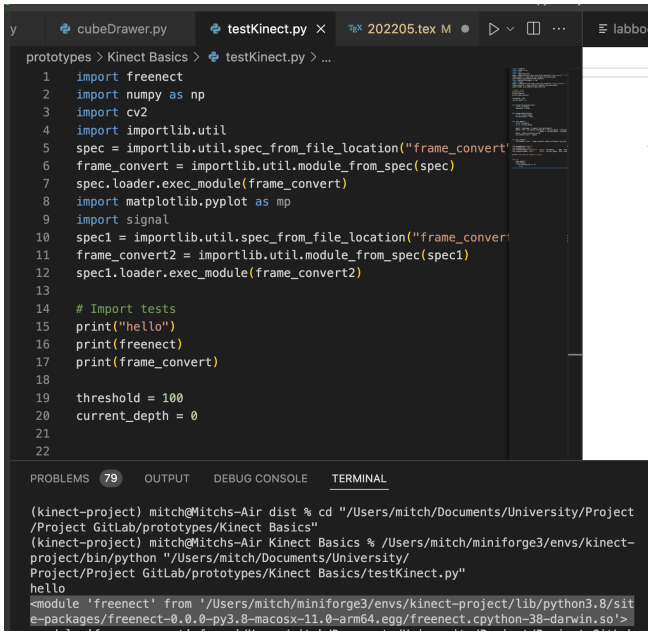


Figure 3.1: Successful output of importing the libfreenect library

Saturday, 07 May 2022

Virtual object control prototype using gesture input

The objection of this session is to construct a barebones prototype that can control a virtual object using gesture control.

The mediapipe hands system is a freely available and excellent hand skeleton recognition system built to take camera input and output a 21-landmark hand model in realtime. This will be utilized as an off-the-shelf solution at first for part of the gesture recognition system. A CDN version of the model is available online and this is integrated into a local HTML, CSS and Javascript project so that a laptop camera can be used for input to the prototype. The prototype HTML file and CDN model can only run on a webserver and not just by opening the HTML file in a browser thus the WebServer For Chrome extension was used to run a local web server to host the application. Code was repurposed from the official mediapipe hands demo program available online. Fig. **[3.2]** shows the prototype application running on a laptop.

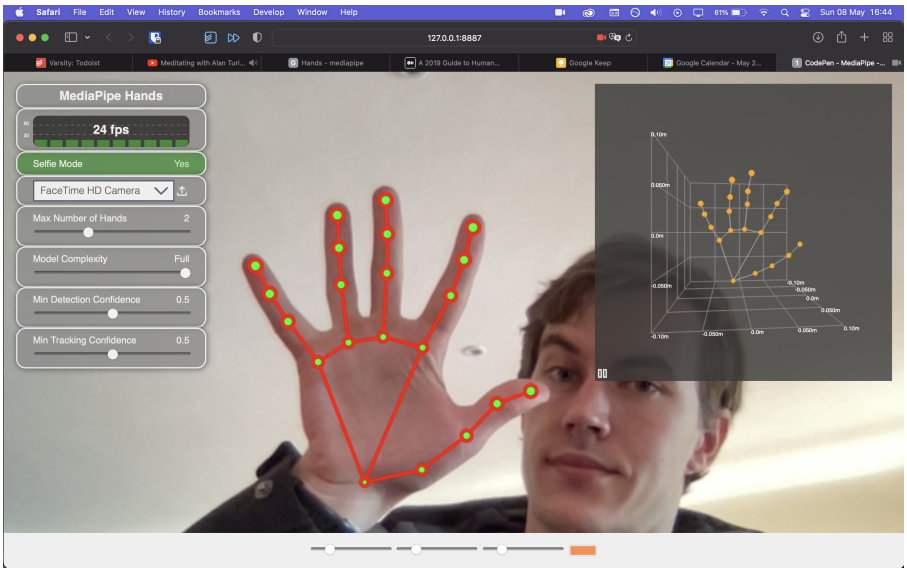


Figure 3.2: Local mediapipe hand model prototype

Mediapipe was not available for Python for the laptop being used for the prototype but the Kinect library available was only available in Python. In order to interface the mediapipe implementation with the Kinect a way of connecting a Javascript and Python program was necessary. To that end, a Python Flask web server was created in a separate Python file to receive information from the hand model. This Flask implementation waited for a PUT request on port 5000 and the Javascript app was set up to send the information for landmark[20] over this connection - the x,y and z co-ordinates for the pinky finger. This was received by the Flask app and then written to a text file. A third program was created in Python to read the values from the text file and draw a cube at co-ordinates matching those of the latest gesture interpretation of the pinky finger over a static image - to be replaced by the feed from the Kinect at a later stage.

The result of this system is presented in Fig. **[3.3]** and shows how moving the user's hand renders the cube in different locations over the image. Because of the multiple files being used to hack together this prototype it runs very slowly and the output is extremely laggy compared to hand input. However, the proof of concept is

shown - a virtual object can be manipulated simply by tracking the position of one of the user's fingers using a webcam as input. Aspects to consider for the fully-fledged implementation from first principles are connecting the system to a depth-sensing camera for depth information of the environment, reducing the number of memory storage and retrieval operations between accepting input data from the hand model and cube rendering steps as well as mapping the hand model input to control of the virtual object better than just tracking a single finger - how the hand moves and how fast should also influence the output imagery.

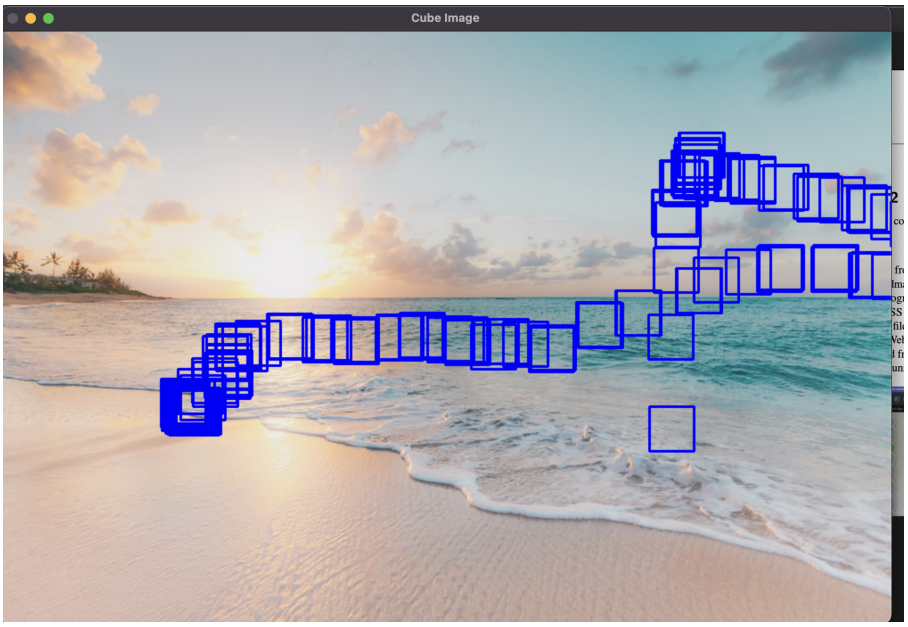


Figure 3.3: Output of mediapipe hand model prototype

In order to run the various files together to get the prototype functioning correctly the following instructions have to be followed.

- Start Chrome web server in dist folder of media pipe folder
- Run flask.py program
- Open index.html of mediapipe folder
- Run cubeRunner.py program
- Move hands in desired manner

Monday, 09 May 2022

Joint regression research

Research was conducted on how the industry standard approach to pose estimation works - the broad subfield that hand gesture recognition falls into. A summary of pose estimation research papers from various institutions points to the general trend of convolutional neural networks being used for linear joint regression - in which the location of particular human joints in various images is slowly learned from labelled data. In particular, the Deep Pose implementation [9] by Toshev et al. uses this approach to great success by stacking a number of layers after each other, alternating whether the layer is a linear or non-linear transformation in a general Deep Neural Network (DNN) model. More research into the details of convolutional neural networks, activation functions and transformations that work well with image data is needed.

Monday, 16 May 2022

The aim of this session is to elucidate the work done on developing a basic neural network in the past few days in addition to experimentation with a PC webcam as input.

Webcam resolution prototype

The webcam of a modern PC is accessible by Python libraries and can be used as input for various programs such as the neural network developed below. Modifying the input resolution of the image is going to be a necessary step in order to train the neural network in a reasonable amount of time and thus a webcam_resolution.py prototype was developed in order to change the resolution of a webcam from which a Python program accepts input. The front-facing webcam on a Macbook Air M1 only allows the resolutions of 1280x720px or 640x480px to be set as the input resolutions. The use of this built-in webcam versus an external piece of hardware will need to be investigated during the training process to see if larger images are too computationally intensive to train and predict with in an online fashion - without extensive preprocessing before training commences.

Neural network prototype

In most of the pose estimation [9] and gesture recognition research papers [8] observed so far the solution ultimately boils down to a fully-connected neural network at the end of the processing pipeline. This is often a convolutional neural network with multiple convolution layers at the start of the network that modify an input image in various ways and then is attached to a fully-connected network at the end, which learns complex patterns in the input image. Thus, the first-principles development of a neural network that can accept various size inputs and train effectively is a necessity for this project.

To that end, the NN.py prototype was constructed which has one input layer, one hidden layer and one output layer of two neurons each. An array of hidden layer biases and output layer biases are also present. These values are easily modifiable to be larger later. The activation function used to activate each neuron after the weighted inputs have been summed is the sigmoid function because of its easily calculable derivative. This too can be changed out for a RELU or tanh function later should the system require it.

The backpropagation algorithm is implemented in the standard method of calculating errors at the output layer based on expected output for a given input, working out the delta values required to modify each output neuron to its expected value and then methodically working backwards through the network to calculate the error and delta at each neuron based on the errors and deltas further forward in the network. The weights between layers are then updated using the delta values, learning rate and input values to that weight. This is done successively for multiple input training data items and the whole process is repeated for a number of epochs in order to train the network successfully. Fig. **[3.4]** shows the rough distribution of the training data and how the neural network needs to learn to differentiate between the two different groups of points. The results of the training process are visible in Fig. **[3.5]** for 600 epochs and the ability of the network to learn is clearly demonstrated.

Similarly in Fig. **[3.6]** the use of more training epochs trains the network more rigorously and allows a lower error rate on testing data. Of note is that the network has only been tested thus far on the same training data it was originally trained with - thus extreme overfitting is no doubt occurring and techniques such as early stopping, regularization and slower learning rates will need to be investigated when training the network for complex tasks like joint regression.

Fig. **[3.7]** shows the output of the first several epochs of training the network using a learning rate of 0.1 and clearly demonstrates the decreasing loss function as the network generalizes to the data given to it. As the

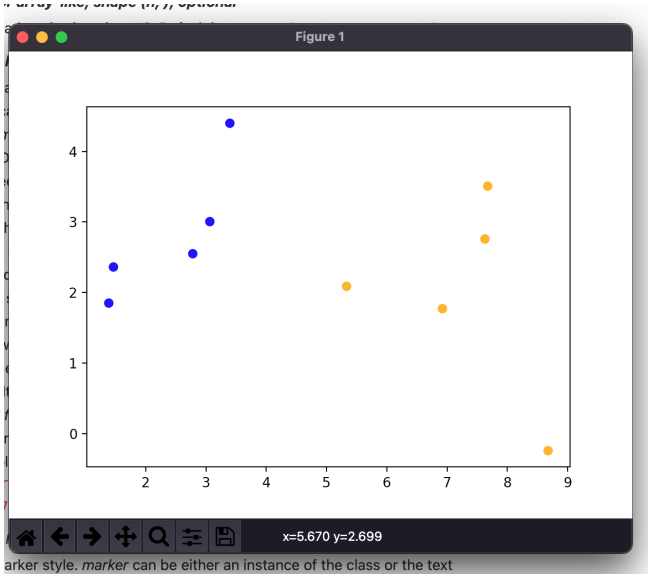


Figure 3.4: NN training data distribution

epochs increase, the loss becomes even smaller and eventually reaches near-zero values and this is visible in Fig. **[3.8]**.

Fig. **[3.9]** shows the output of a testing regime that predicts the values for all the training data inputs and achieves a 100% accuracy for the testing.

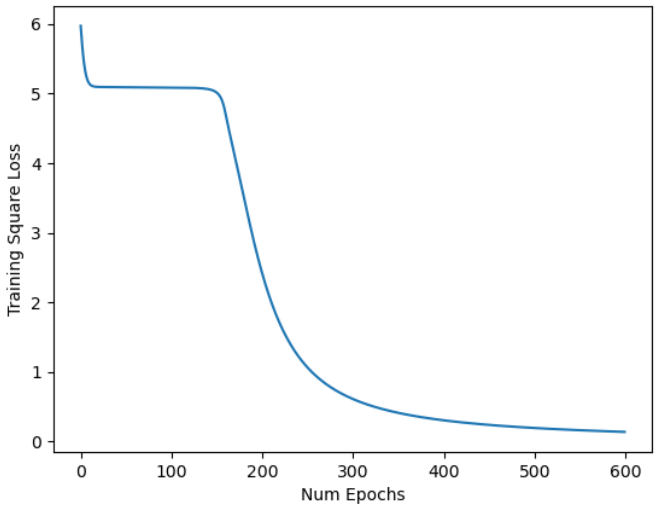


Figure 3.5: NN prototype loss curve over 600 training epochs

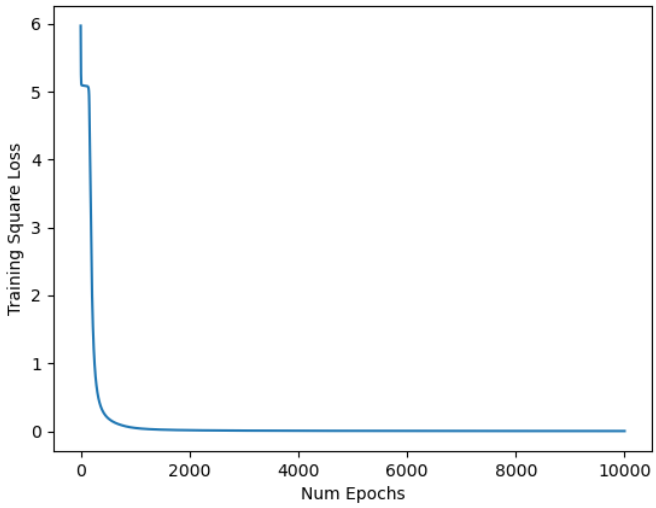


Figure 3.6: NN prototype loss curve over 10000 training epochs

```
Epoch:  0, Learning rate:  0.1, Error:  5.97404275605497
Epoch:  1, Learning rate:  0.1, Error:  5.796521409271948
Epoch:  2, Learning rate:  0.1, Error:  5.642140128576829
Epoch:  3, Learning rate:  0.1, Error:  5.512424949971091
Epoch:  4, Learning rate:  0.1, Error:  5.407095253662301
Epoch:  5, Learning rate:  0.1, Error:  5.324252798605831
Epoch:  6, Learning rate:  0.1, Error:  5.260907697706434
Epoch:  7, Learning rate:  0.1, Error:  5.213609367316193
Epoch:  8, Learning rate:  0.1, Error:  5.1789709646896265
Epoch:  9, Learning rate:  0.1, Error:  5.153993725250735
Epoch:  10, Learning rate:  0.1, Error:  5.136202855487615
Epoch:  11, Learning rate:  0.1, Error:  5.123654260718681
Epoch:  12, Learning rate:  0.1, Error:  5.114873219367095
Epoch:  13, Learning rate:  0.1, Error:  5.1087686582287155
Epoch:  14, Learning rate:  0.1, Error:  5.10454776888251
Epoch:  15, Learning rate:  0.1, Error:  5.101642049551113
Epoch:  16, Learning rate:  0.1, Error:  5.099647907210916
```

Figure 3.7: NN prototype training output over 600 training epochs with learning rate of 0.1

```
Epoch:  587, Learning rate:  0.1, Error:  0.14303256325699895
Epoch:  588, Learning rate:  0.1, Error:  0.14257718717405177
Epoch:  589, Learning rate:  0.1, Error:  0.14212398062595982
Epoch:  590, Learning rate:  0.1, Error:  0.14167292572650256
Epoch:  591, Learning rate:  0.1, Error:  0.1412240047714976
Epoch:  592, Learning rate:  0.1, Error:  0.14077720023671897
Epoch:  593, Learning rate:  0.1, Error:  0.14033249477585127
Epoch:  594, Learning rate:  0.1, Error:  0.13988987121847646
Epoch:  595, Learning rate:  0.1, Error:  0.13944931256809398
Epoch:  596, Learning rate:  0.1, Error:  0.13901080200017277
Epoch:  597, Learning rate:  0.1, Error:  0.13857432286023602
Epoch:  598, Learning rate:  0.1, Error:  0.13813985866197687
Epoch:  599, Learning rate:  0.1, Error:  0.13770739308540492
```

Figure 3.8: NN prototype training output end over 600 training epochs with learning rate of 0.1

```
Expected:  [0, 1] , Got:  [0.01319445 0.98642943]
Expected:  [0, 1] , Got:  [0.0077593  0.99163586]
Expected:  [0, 1] , Got:  [0.00693049 0.99242094]
Expected:  [0, 1] , Got:  [0.00974547 0.98971331]
Expected:  [0, 1] , Got:  [0.01013226 0.98939783]
Expected:  [1, 0] , Got:  [0.99072868 0.0097442 ]
Expected:  [1, 0] , Got:  [0.98680139 0.0134023 ]
Expected:  [1, 0] , Got:  [0.99106311 0.00943854]
Expected:  [1, 0] , Got:  [0.9913314  0.00919796]
Expected:  [1, 0] , Got:  [0.98880203 0.01157191]
```

Figure 3.9: NN output of overfitting test on small input dataset

Tuesday, 17 May 2022

Kernel Functions Experimentation

The aim of this session is to investigate the purpose and ease of using kernel functions in convolutional networks.

In the research papers studied so far the use of convolutional neural networks is prevalent for taking image input and acquiring a desirable output result using a neural network. The use of these convolution layers involves multiplying the input image pixel values with various kernel functions that modify the output of the image in such a way to generalize certain regions of an image and capture nonlinear components present in the image. These kernel functions can be applied in numerous ways, including to individual RGB layers, to sums of the layers and to different parts of the image. A kernel function prototype was built in order to attempt to use these kernel functions on input images and acquire a sense of the function they perform and how they assist in a convolutional neural network's learning.

Fig. **[3.10]** shows the input image that is passed into the various kernel functions. The first kernel is the edge detection 1 kernel and is presented in Fig. **[3.11]**. The rough edges of the person in the input image are visible using this kernel. A more prominent and striking interpretation of the edges of the image's content is visible in Fig. **[3.12]**. Two alternative kernel functions are presented in the Left and Right Sobel kernel functions respectively in Fig. **[3.13]** and Fig. **[3.14]**. The utility of these kernel functions is apparent in that they extract meaningful information from an input image and allow reduced computation to occur in a neural network while still representing salient features of an input image.

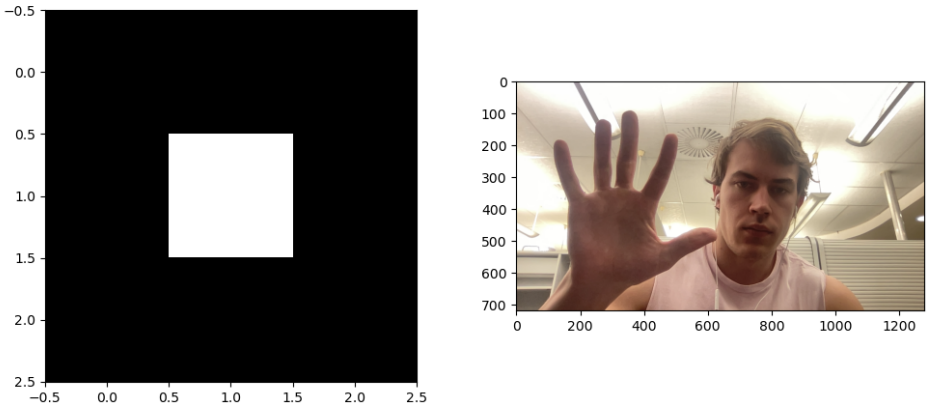


Figure 3.10: Input image to various kernel functions

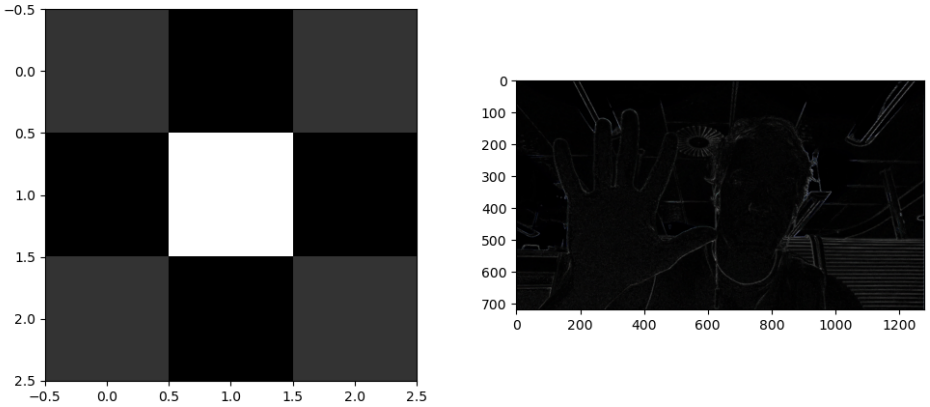


Figure 3.11: Output of the edge detection 1 kernel function

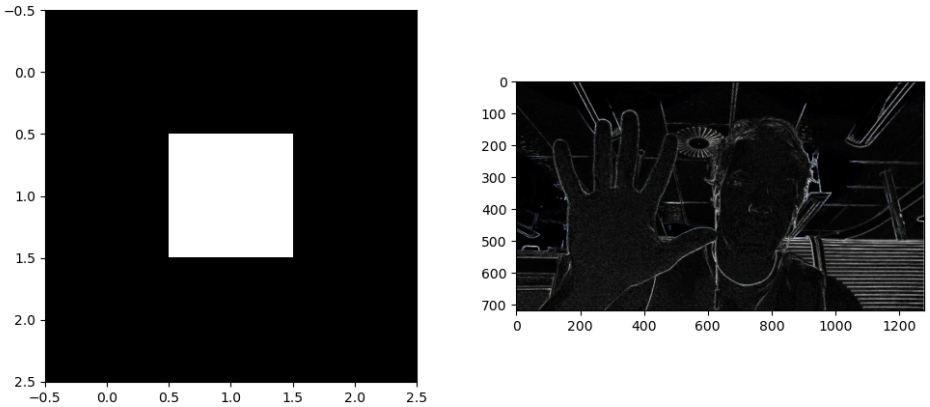


Figure 3.12: Output of the edge detection 2 kernel function

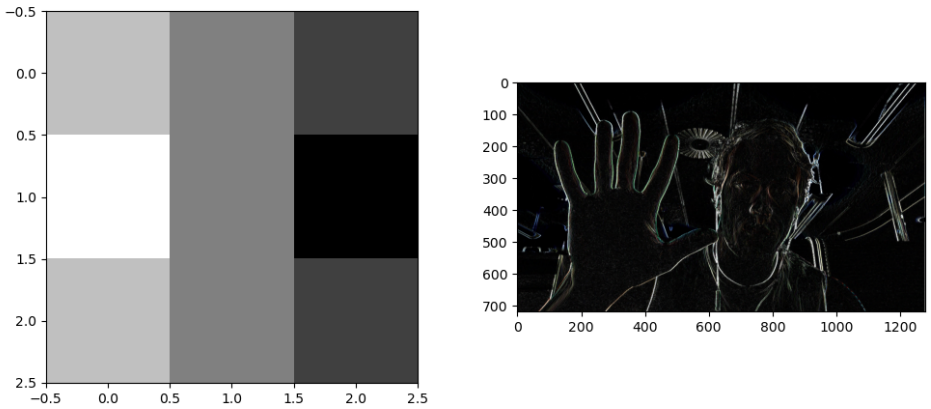


Figure 3.13: Output of the left sobel kernel function

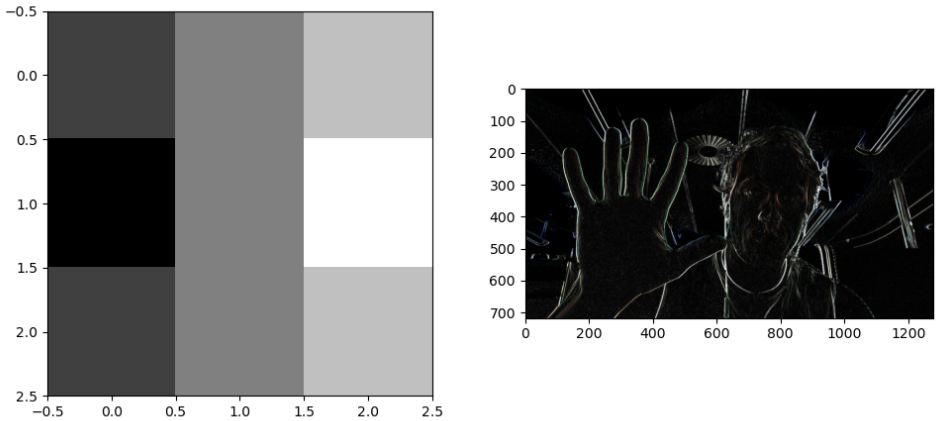


Figure 3.14: Output of the right sobel kernel function

Tuesday, 31 May 2022

OpenGL Cube Experimentation

The aim of this session is to illustrate the work done in the previous week on the three-dimensional rendering of a cube in a video stream.

In order to properly create an augmented reality application the virtual object must be rendered continuously alongside a changing input video stream with different transformations and perspectives in order to create augmented reality. The rendering of this virtual object will be accomplished using a library. OpenGL is the de-facto standard for rendering three-dimensional scenes on modern hardware and it possesses a computationally efficient Python wrapper API. This API was used to render a basic cube as seen in Fig. **[3.15]**.

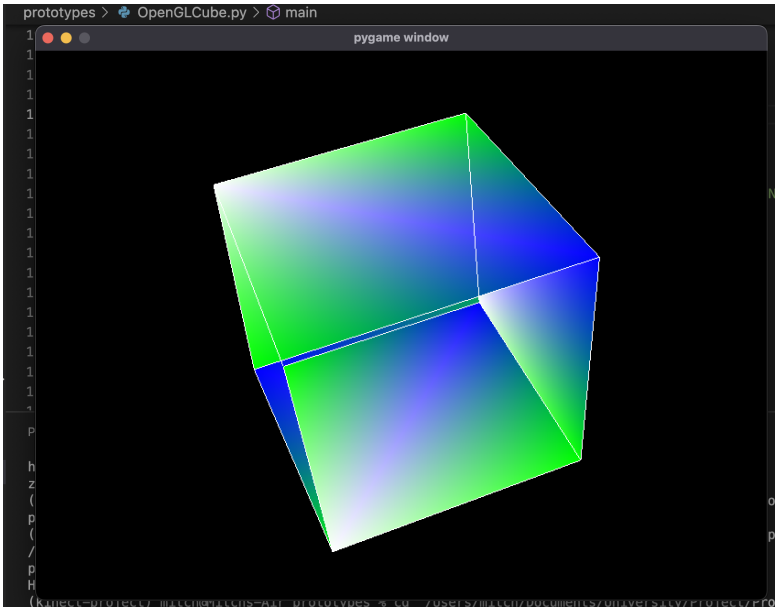


Figure 3.15: Output of the OpenGL cube rendering program

OpenGL utilizes a series of vertices and lines between these vertices in order to draw three-dimensional shapes. The cube obviously consists of these primitive shapes but the background too is merely a giant rectangle that stretches the length of the screen and has mapped to it the texture of the input video feed - broadcast to it from a webcam. This effect is visible in Fig. **[3.16]**. Further work needs to be done to investigate the rotation of objects in the OpenGL application as well as perspective changes needed to make the object appear smaller and larger at different times.

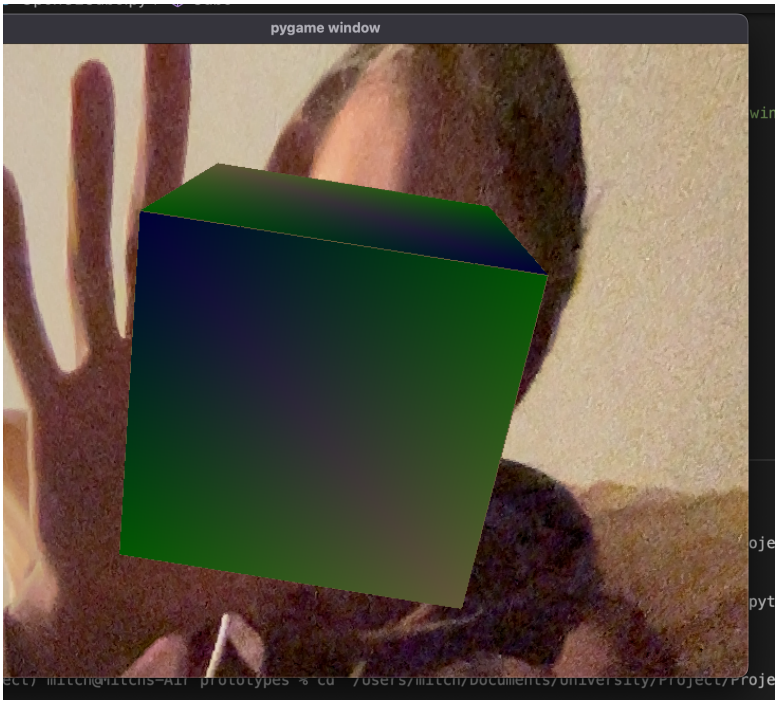


Figure 3.16: Output of the OpenGL cube rendering program overlaid on a video input feed

4

June 2022

Monday, 06 June 2022

Further OpenGL Prototyping

This session focuses on improvements made to the initial OpenGL cube rendering prototype.

Further experimentation was performed with the OpenGL library in order to rotate, translate and adjust the virtual object or cube in a desired manner. The **glTranslatef(x,y,z)** and **glRotatef(angle,x,y,z)** commands were used to affect the cube in various ways. The perspective system of OpenGL only allows the "camera" or user's perspective to be shifted - not individual objects within the scene. Thus thinking about how to construct the virtual object control system relies upon figuring out how to move the perspective of the camera or point of view as the user provides gesture input. This is further complicated by the fact that the background imagery of the user or scene is merely projected onto an orthographic quad rendered at the back of the OpenGL scene. Thus the idea of a far and near clipping plane must be kept in mind when moving objects between different x,y and z co-ordinates. Getting this background quad rendered and not also moving when the perspective and translation changes are made took many hours of troubleshooting.

The completed prototype visible in Fig. **[4.1]** allows a user to move their hand and the system tracks the location of their pinky finger and attempts to adjust the perspective based on the position of the pinky finger within the camera's frame of view. Further experimentation is needed to adjust the cube at different speeds and produce more realistic output.

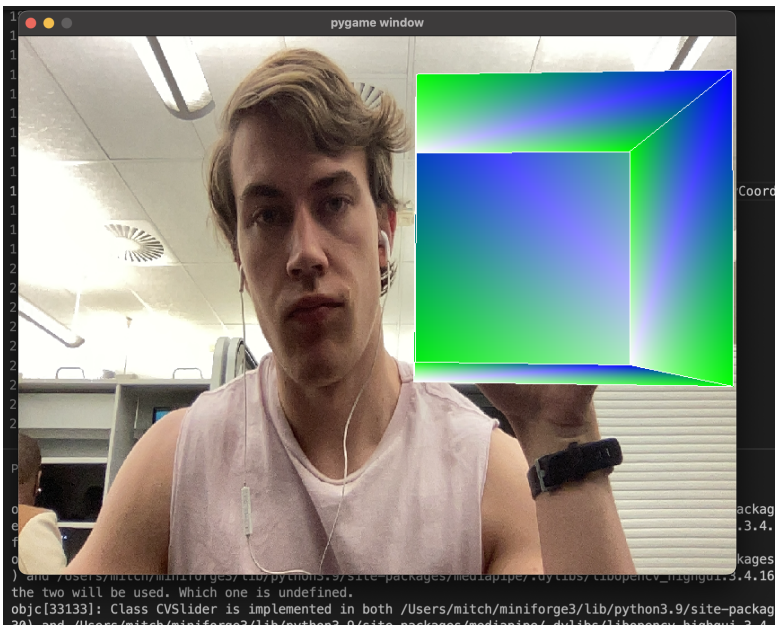


Figure 4.1: Output of the OpenGL and Mediapipe hands prototype application

Tuesday, 07 June 2022

OpenGL And Mediapipe Prototyping

In order to arrive at the end of the semester with a working prototype assembled from prebuilt components, the integration of Mediapipe Hands and the OpenGL cube rendering program was undertaken and its implementation detailed here.

Mediapipe hands has been used in earlier prototypes for hand detection and will again be used now to generate a three dimensional model of the user's hand from webcam input. Earlier prototypes utilized the Javascript version of mediapipe hands since the Apple Silicon API could not be installed but after much trial and error this was rectified and simplified the process a lot - a single Python file could now be written that combines the OpenGL cube rendering prototype with the mediapipe hand detection model that tracks the co-ordinates of a user's fingers and utilizes it as input for the cube's position on the screen. The output of the mediapipe Python example file can be seen in Fig. [4.2].

Figure 4.2: Output of the Mediapipe hands python example application

A video demonstration of the prototype's functioning can be found in the Project's repository under Prototypes but the system was able to accurately take the location of the user's pinky using mediapipe and the code

```
pinkyCoords = hand_landmarks.landmark[20]
```

and use that as input to the OpenGl translate and rotation commands which scaled and adjusted the cube in order to try and be placed on top of the user's hand. This was accomplished with

```
glTranslatef(-(lastX-pinkyCoords.x)*1.5, -(lastY-pinkyCoords.y)*1.5,
    -(lastZ-pinkyCoords.z)*1.5)
glRotatef(1, -(lastX-pinkyCoords.x)*1.5, -(lastY-pinkyCoords.y)*1.5, 0.0)
```

PyGame was utilized to display the OpenGL rendering as well as mediapipe hands regression output. Further refinement is needed to move the cube at the speed the user moves their hand and to be closer to the user's hand rather than further away. Depth information from the Kinect sensor also needs to be considered in order to create a synthetic representation of the environment that can be compared against and used to perform collision

avoidance and realistic physics additions to the prototype such as the mimicking of gravity or outside influences on the cube.

Friday, 17 June 2022

Project Proposal Revision

The aim of this work session is to revise the project proposal according to Prof. Hanekom's feedback on the first submission and to clarify certain aims of the project with regards to the processing platform it will be conducted on.

The main area of concern regarding the first proposal submission was the processing platform. As per an email dated Friday 17th June with Mr. Grobler (see Appendix) it was determined that an embedded platform should be used in the project and that "certain aspects be demonstrated on the platform." However, it is expected that in order to achieve the necessary requirements and specifications outlined in the proposal that a PC platform will have to be used. Following from this, it was decided to update the proposal with the use of the embedded platform as the main processing platform but specify in the implementation section that the main challenge of the project will be to achieve the system's functioning on an embedded platform due to the computationally intensive nature of the gesture control algorithm and virtual object rendering.

It is expected that an embedded platform that falls within the three-thousand rand budget of the project will be unsuitable to achieve the performance described in the specifications as they were initially developed in the first proposal submission assuming a PC platform would be used for processing. It is hypothesized however that just the gesture recognition algorithm could be optimized to the point where it could run effectively and meet specifications on a standard embedded platform like a Pandaboard or Odroid. The gesture recognition system may also be able to run slowly on the same platform with a much reduced frame rate. The use of two embedded platforms in tandem - one running the gesture recognition system and the other the virtual object rendering could be a possible way to achieve the performance specifications on an embedded platform but it is doubtful if this could be accomplished within the budget.

The following additional points of feedback were given about the proposal. The incorrect fonts were used in the proposal and this has been rectified by adhering more strictly to the proposal template. Requirement 6 of the system requirements and specifications was marked as a redundant specification. This specification stated that 9 known gesture templates had to stored in memory - this was implied earlier by requirement 2 where the system was expected to be able to recognize 9 discrete gestures. The requirement has thus been removed. Requirement 1 stated that user's hand gestures and corresponding changes to the position and orientation of the virtual object would have a latency of less than 41.6ms. However, it was not explained why this was the case - standard filmmaking techniques utilize standard frame rates that update the picture displayed on a screen 24 or 30 frames per second.

Values less than 24fps make video footage appear choppy or disjointed to the human eye because below this value the brain struggles to simulate "fluid motion." Thus it is necessary to use at least 24fps for the virtual object rendering system and gesture recognition system otherwise the illusion of real-time virtual object control will be destroyed. Higher frame rates than 24fps will appear to make the virtual object move more smoothly but would require a corresponding increase in the amount of inferrences for the hand gesture detection system and since the embedded system is already expected to have difficulties meeting the 24fps specifications it was decided that this should remain the target refresh rate of both the virtual object rendering algorithm and gesture detection algorithm. The motivation in requirement 1 was updated to reflect this reasoning.

In the description of what the off-the-shelf components of the system would be, there was concern in requirement 1 about what constituted "video processing libraries" as only "basic low-level operations may be done with the assistance of libraries." The requirement was updated to reflect that the only libraries to be used off-the-shelf will be those used for video capture, image to array conversion and image display to the user. Finally, in the Design and Implementation Tasks section of the proposal, the use of the imperative form to describe individual design and implementation tasks was criticized - the grammar of this section has thus been rectified to remove use of the imperative form.

In conclusion, the system is expected to function on an embedded platform but with severe performance deficiencies due to the computationally straining nature of rendering 3D video alongside a system that can accept two streams of video input and perform gesture recogniton on one of those streams of input. Conducting only the gesture recognition computations on the embedded platform is expected to be doable and the potential use of multiple embedded platforms is a potential way of fulfilling the project specifications in their new form.

5

July 2022

Wednesday, 06 July 2022

Convolution and max pooling algorithms

The aim of today's session is to implement the convolution and max pooling algorithms and interface them with the basic neural network already implemented. The logic of the convolution algorithm is presented in Fig. **[5.1]** and shows how the algorithm will be implemented from first principles.

The convolution algorithm is initially constructed with stride length one and zero padding and accepts an input and filter matrix. A counter iterates over the entire input matrix and another counter sums all the multiplications between the current portion of the input matrix and each element of the filter. This sum is then placed into the first position in an output matrix. The intuition and calculations described here were first evolved from the calculations shown in Fig. **[5.2]**.

The max pooling algorithm has a modifiable pool length and stride length and accepts an array of neurons that represents the layer of the network to be max pooled. Similarly to the convolution algorithm, a counter iterates over the entire previous layer and considers each pool-length by pool-length portion of the previous layer and finds the maximum value in this portion of the matrix. This maximum value is placed in an output matrix.

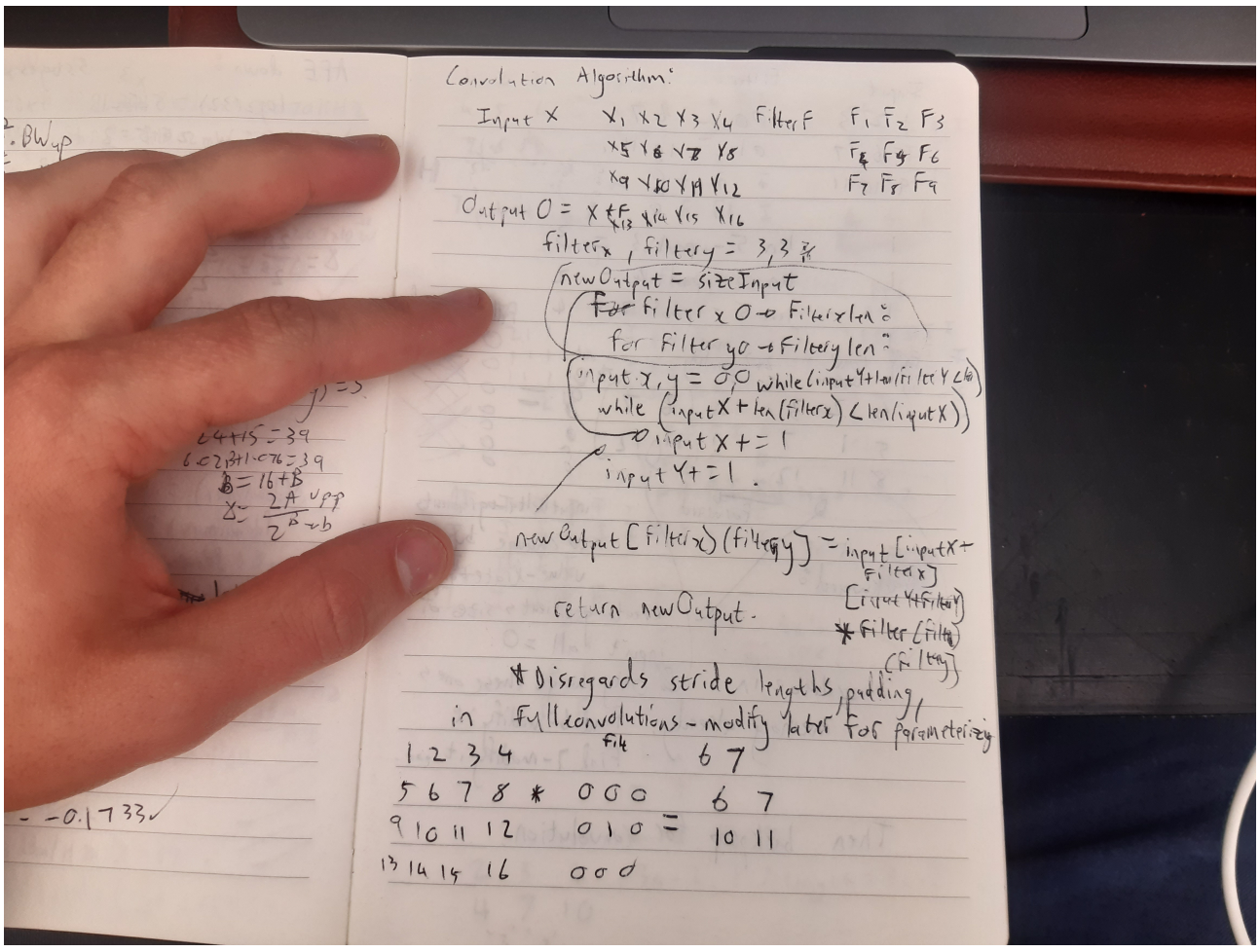


Figure 5.1: Proposed logic for convolution algorithm

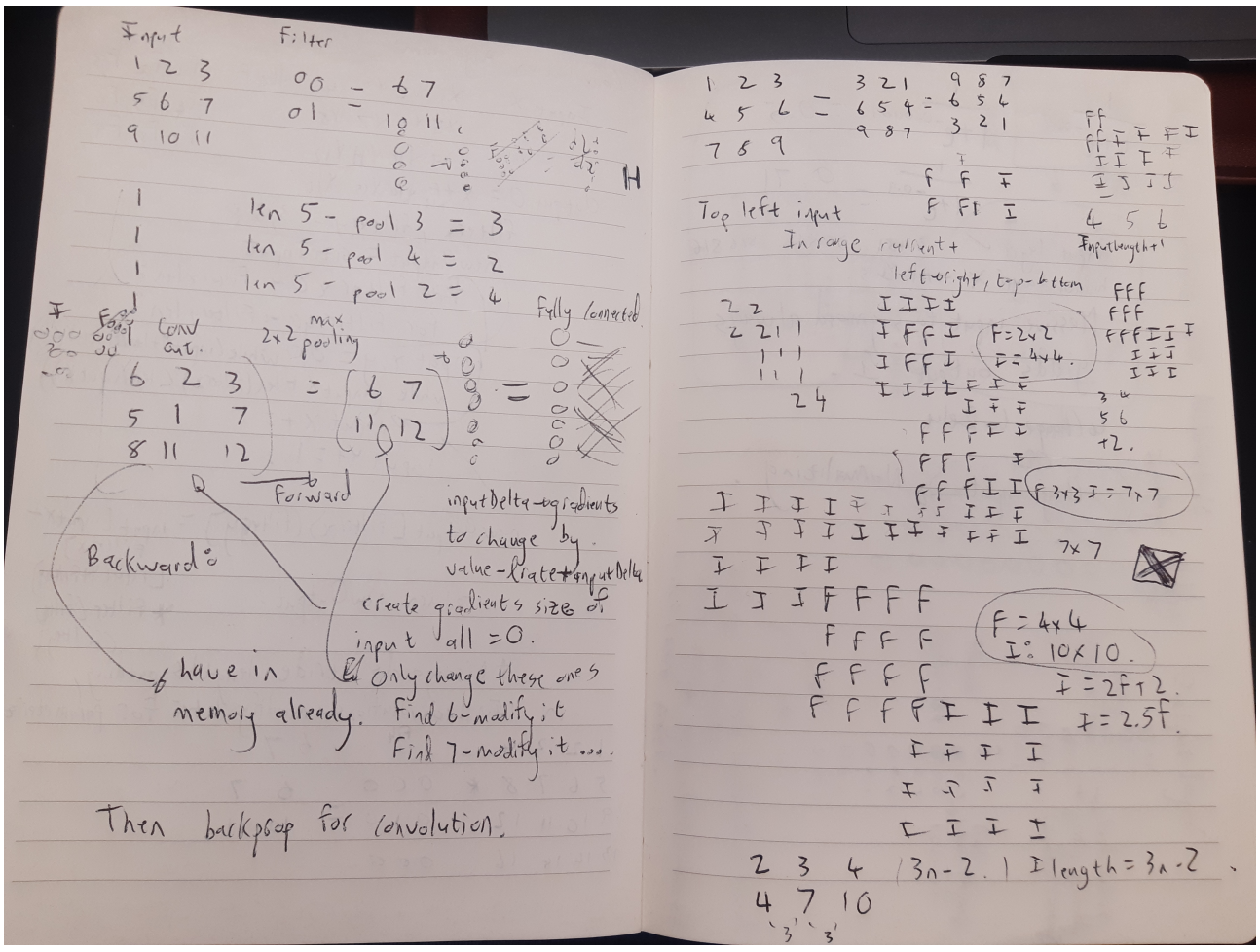


Figure 5.2: Calculations and logic for convolution algorithm

Thursday, 07 July 2022

CNN backpropogation for convolutional layers

Further work was conducted today on the max pooling algorithm and implementing the backpropagation of the convolutional layers - this involved finding the gradient of each filter element based on the error backpropogated from the fully-connected neural network to the max-pooling layer. The max-pooled output is found inside the convolutional layer and the gradient applied to this specific part of the filter by performing the convolution of the input image data and the error from the previous layer.

Friday, 08 July 2022

Basic CNN testing

A single image was inputted into the convolutional neural network today in order to establish that the network could overfit and that the backpropagation process was working correctly. The backpropagation algorithm was also modified so that it did not crash upon running - this was due to incorrectly-shaped hidden and output layer gradients being created.

Tuesday, 12 July 2022

Troubleshooting exploding weights and saturating neurons

The past several days were spent troubleshooting the neural network and trying to establish why the weights of the network were getting larger and larger and why all the neurons in both the hidden layer and output layer were all saturating at a value of 1.0. After printing out all of the values of the neurons in each layer it was realized that the sigmoid activation function was outputting the 1 value for all its inputs because the initial input to the network was not normalized and was just the RGB value of each pixel ranging from 0-255. After normalizing the input by dividing by 255 so a value between 0 and 1 was presented as input the problem seemed to be mostly rectified. Further modifying the initial values of the weights to be randomly set to between -1 and 1 improved the network training a lot and prevented the saturation of all neurons in the hidden and output layers. The learning rate was changed to 0.001 from 0.1 and although the network took much longer to train the loss of the network improved a lot - down from around 8 to 0.01 for one input image.

Successfully overfitting one image was the initial goal of building the network and the work session today went far towards achieving that goal. Furthermore, the sigmoid activation function was swapped out for a relu activation function but the exploding weights and saturating neurons value became an issue again - further troubleshooting of the forward propogation process might be necessary since the industry standard for activation functions in convolutional neural networks working with images is the relu function. Normalizing the output of each neuron before activation might be a necessary step. Another parameter that needs to be tweaked in future is the stride length used in the convolution operation - how far along the input matrix to move the filter matrix for each multiplication in the convolution operation. The mathematics of the convolution operation however have proven difficult to get right and troubleshoot so this is left for a future work session.

Wednesday, 13 July 2022

Output debugging in a textfile

The aim of the work session today was to enable better output debugging of the network. To that end, a writeToTextFile method was created that could accept strings and integers and then output those values to a textfile in the same directory as the network that was erased and updated every time the network was run. This enabled the full visualization of all the neurons in the hidden and output layer as well as what the values of the filters are - thus making it possible to identify that the filters were not changing as they should be - the problem being that with only one input image the value of the filter did not really need to change as the fully connected neural network was changing enough to reduce the loss at the output of the network. Adding more training images rectified this problem. Additionally, the full convolution required in the backpropagation of the convolutional layers was implemented using the scipy convolution2d method to be replaced by a first principles implementation in future.

Thursday, 14 July 2022

MNIST hand-written digit classifier

The aim of the work session today is to modify the convolutional neural network in order to create a MNIST hand-written digit classifier just to confirm that the network can learn and predict image values correctly. Ten input images were downscaled to 28x28 and fed into the network. The output layer was modified to only have 10 output neurons while the hidden layer has 1000 neurons and the network was trained over 300 epochs - enough to get the loss down below 1.0. Fig. **[5.3]** shows the output of the first five input images to the MNISt classifier which are images of the handwritten digits 0-4. The output shows that the largest value (highest probability) in each test case is the correct value - the first neuron for the input image showing a 0 and the second neuron for the input image showing a 1. To make it easier to understand the output, the final layer of the network should be a softmax layer which just outputs the number neuron with the highest probability in the output layer. Thus the CNN can classify handwritten 28x28 digits.

```
Expected:  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] , Got:  [7.83798092e-01 2.19948277e-08 9.11235169e-02 9.91394030e-03
 1.63857517e-04 6.12500441e-05 1.32994910e-02 3.18618763e-02
 2.15026227e-08 1.64269077e-01]
Expected:  [0, 1, 0, 0, 0, 0, 0, 0, 0, 0] , Got:  [2.21830494e-02 8.88950656e-01 2.53807643e-05 9.91588825e-02
 1.04559852e-08 2.92263183e-02 3.67727980e-03 6.41452326e-08
 2.31667143e-03 3.66354380e-03]
Expected:  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0] , Got:  [1.95853017e-02 2.66851399e-10 8.53764525e-01 6.43753871e-03
 3.89665206e-05 3.76966484e-06 1.32812006e-02 1.72386132e-02
 1.78595957e-08 6.44644551e-02]
Expected:  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] , Got:  [7.17439460e-02 1.61142613e-08 2.65698453e-02 8.07973342e-01
 5.18160821e-06 4.35058145e-03 8.31315029e-02 3.66486451e-05
 2.60789939e-07 1.68402402e-04]
Expected:  [0, 0, 0, 0, 1, 0, 0, 0, 0, 0] , Got:  [3.30205417e-02 4.64639109e-02 3.16977413e-06 6.91976800e-02
 9.84533678e-01 8.66347388e-03 1.77145881e-02 1.65967066e-03
 2.05444713e-05 1.35929615e-06]
```

Figure 5.3: First five output results of the MNIST classifier

The network however took over 6 seconds to run each epoch and as a testing scenario lead to the concern about the network not being optimized enough for realtime use - especially later when the network will need to predict hand positions over 24 times a second. Thus it is going to be necessary to optimize the network, rewrite the convolutional operation as well as optimize processes like the max pooling if the network is going to perform adequately at a later date.

It is noted too that the revised project proposal was received from Prof. Hanekom today and the proposal was accepted. The accepted proposal has been placed in the proposal folder in the project repo and now stands as the standard with which to measure progress and the project's capabilities against.

Saturday, 16 July 2022

Single hand coordinate classifier

Time was spent the previous two work sessions on saving the weights of the CNN to a .npy file for easy re-use and to enable "trained" networks to be used again without having to repeat the whole training process everytime the network was initialized. The MNIST classifier was modified to instead attempt to predict the location of a single hand coordinate when given an input image of a hand. The principle design decision with regards to this was whether to make the output layer a collection of neurons that represented the prescence of a hand landmark at a specific pixel (thumbInPixel1,thumbInPixel2...) or to have the output layer a collection of neurons that represented the normalized xy coordinate value of each hand landmark (thumbXPosition,thumbYPosition,pinkyXPosition...) The proposed architecture for the hand coordinate convolutional neural network architecture is presented in Fig. [5.4].

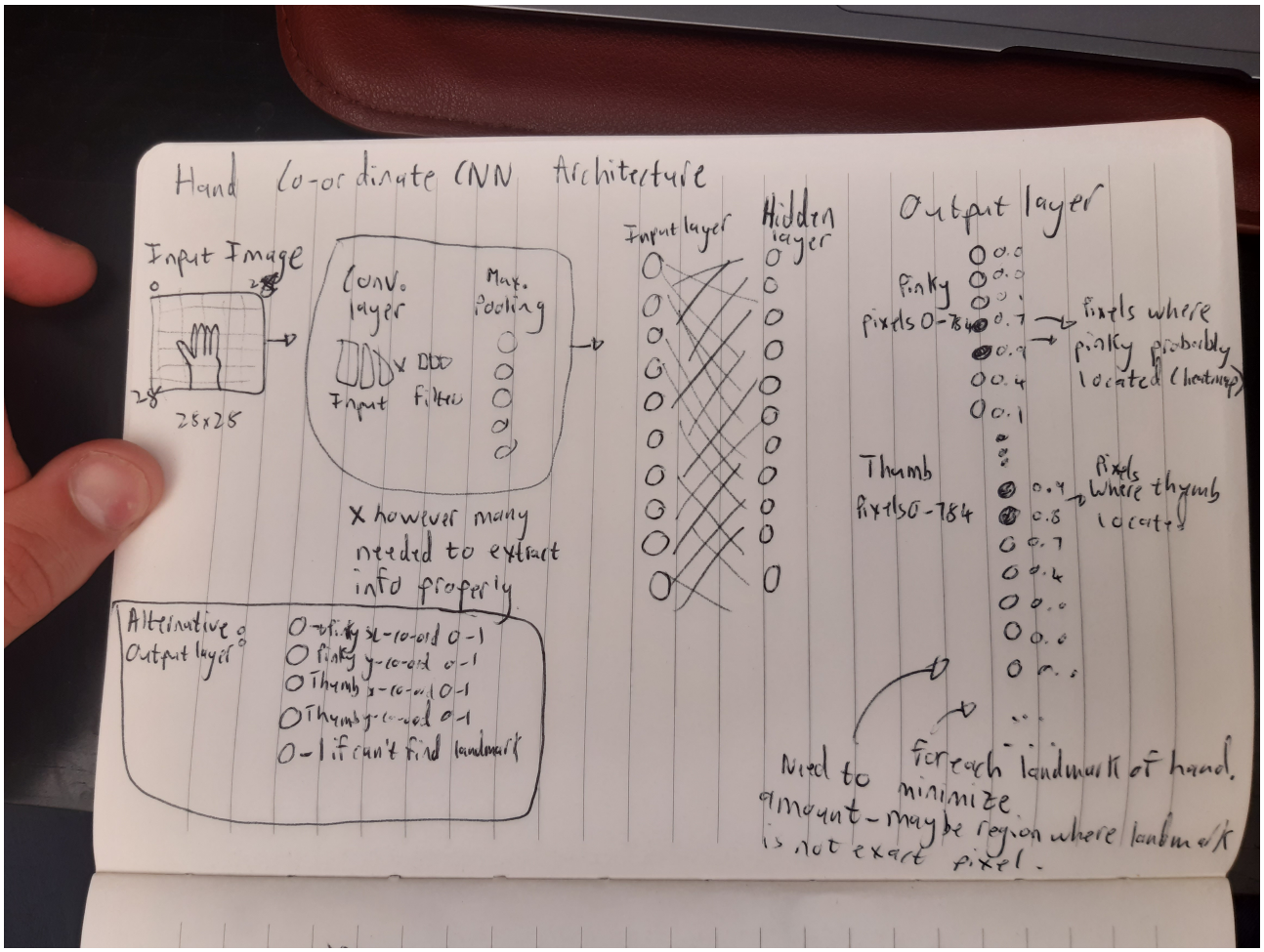


Figure 5.4: Proposed architecture for hand coordinate convolutional neural network

The second option was the first to be attempted because it is so much less memory intensive and the coordinates were set up from 0-1 for both the x and y dimension starting at the bottom left of the input image. The loss of the network was very high and did not seem to be able to classify a single digit. It was postulated that the use of only a single convolutional and max pooling layer was not enough to extract meaningful information

from the input image. Thus time was spent adding a second convolutional and max pooling layer. This work was continued on the 18th of July when the backpropagation method was modified to account for the new convolutional layer.

Monday, 19 July 2022

Gesture classifier

The aim of today's work session is to build a system that can accept hand landmark coordinates and predict the gesture that is represented by those coordinates of the hand.

This system will be implemented with a neural network that takes in 42 input parameters (the x and y coordinates of 21 hand landmarks) and output 9 nine numbers that represent the probability that the hand coordinates represent the nine gestures: "one", "two", "three", "four", "five", "fist", "peace", "rock on" and "okay." These gestures were decided upon to maximize the difference between their physical representations and because the natural "up" and "down" gestures of pointing a finger up or down is easily misinterpreted by the network if the image is merely flipped upside down. 1 hidden layer with 100 neurons is used and the network is trained on 45 input images (5 images for each gesture).

The loss curve for the learning process is presented below in Fig. [5.5]. Some examples of the system accurately classifying gestures are presented in the figures below. The system used mediapipe hands to provide the hand landmark coordinates to the gesture classifier as this has not been accurately completed from first principles yet. The system however does not function excellently and when the hand gesture is presented in different orientations the system cannot classify it properly. This can be remedied with more training data showing the gestures in various orientations and a higher number of neurons in the hidden layer. This will be attempted at a later date when higher accuracy is required.

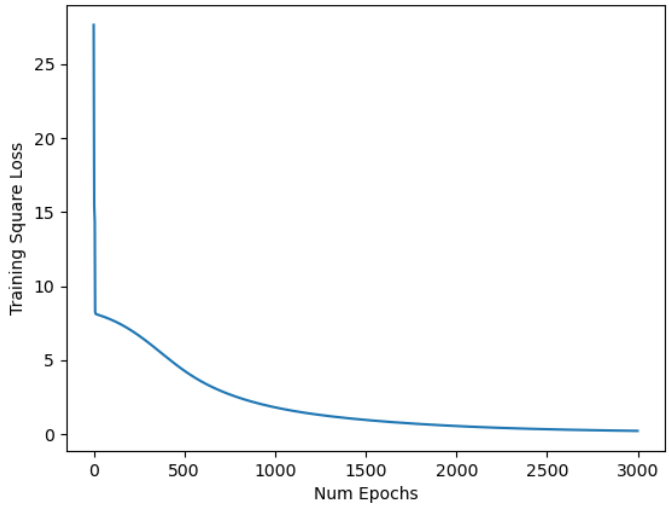


Figure 5.5: Loss curve of the gesture recognition neural network

Figure 5.6: Predicted gesture using gesture recognition neural network

Figure 5.7: Predicted gesture using gesture recognition neural network

Figure 5.8: Predicted gesture using gesture recognition neural network

Figure 5.9: Predicted gesture using gesture recognition neural network

Monday, 25 July 2022

Kinect camera edge detection

The aim of today's work session is to experiment with convolving the images from the Kinect camera with various filters in order to detect edges of objects presented to the camera - in aid of detecting objects to be interacted with in augmented reality by the virtual object.

A prototype was constructed to take the input of the Kinect camera and convolve it with a modified edge detection filter so that only the outlines of shapes and objects in the camera feed are visible in the output of the program. The result is displayed in Fig. **[5.10]**. This method fails to take advantage of the Kinect depth information however and only shows the edges of objects in two dimensions, not three - thus an alternative approach using the depth data needs to be devised to find the edges of objects in three dimensions. Perhaps a combination of the depth data at a relevant edge pixel will be able to inform an algorithm if an object can be moved through that particular coordinate in virtual space.

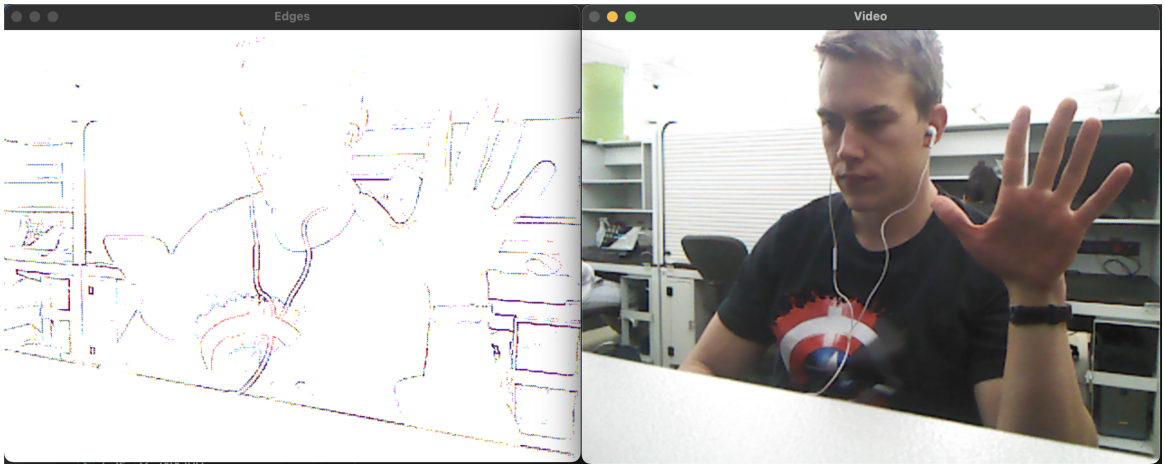


Figure 5.10: Output of convolution of Kinect imagery with modified edge detection filter

It is hypothesized that the final implementation of a collision avoidance algorithm will need to have the following attributes in order to complete its function of integrating the virtual object with the real environment:

- Each xyz pixel in the virtual space must either be an edge or non-edge of an object
- Depth data at each pixel must be available to determine how far back that object issue
- The virtual object must also have xyz coordinates
- The virtual object xyz coordinates must be checked against the environment's xyz coordinates to prevent collisions

It is noted that the depth output from the Kinect camera is an integer value from 0-2048 where 2048 represents a depth value closest to the camera and 0 a value far away from the camera. This can be taken advantage of in order to display the different values using a colormap to represent depth. The results of this process is visible in Fig. **[5.11]**. Thus in order to conduct collision avoidance a three-dimensional coordinate system can be created that is x=camera width, y=camera height and z=0-2048 depth dimensions large and a set of coordinates using this same scale that maps the virtual object's position in the environment to the coordinate system outlined here.

A conversion algorithm will have to be created in order to map the coordinate system's position of the virtual object into OpenGL coordinates for rendering. It is also going to be necessary to shrink the apparent size of the cube as it is moved around the screen in order to create perspective and the illusion of augmented reality.

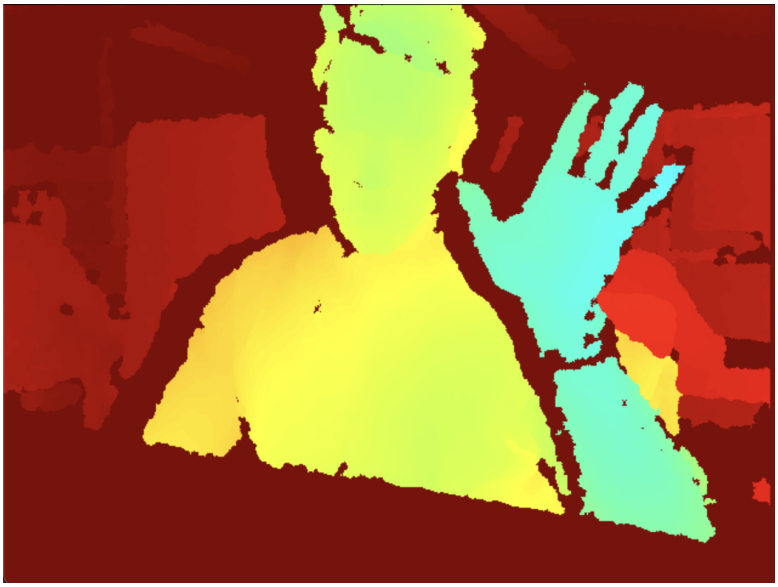


Figure 5.11: JET colormap interpretation of Kinect depth data

A large amount of time today was also spent converting the neural network architecture developed so far into a more modifiable and expandable format using classes and methods - this will enable layers to be added and removed far more easily and will increase readability and understanding of the network. As of the end of the work session, this has mostly been accomplished save for the backwards propagation of the various layer types and forward propagation of the convolutional and maxpooling layers.

Tuesday, 26 July 2022

Neural network refactoring

The aim of the work session today is to continue refactoring the convolutional neural network code for better testing and expansion. After many hours of examining convolutional neural network theory, the network was optimized to allow multiple filters to be used in each convolutional layer - a feature that had been overlooked in the previous implementation and has now been rectified. The use of several convolutional layers in succession however is still non-functional due to the layer being coded to only accept input images with three channels - not multidimensional arrays which are the output of previous convolutional layers. This problem was worked on for many days after this work session and served as a major source of frustration.

6

August 2022

Monday, 01 August 2022

First draft of literature review for first-semester report.

With an increase in the proliferation of powerful personal computing hardware it has become feasible to create augmented reality applications that merge together virtual objects and a user's environment. Similarly, modern computer systems are capable of performing real-time machine learning inference on a large range of inputs and returning meaningful results – this has led to the advent of human-control inputs to computers like hand gesture control.

These two subfields, 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 **Australiaspiders** who utilize augmented reality and gesture control to treat agoraphobia using an interactive augmented reality application as well as **markerlessar** who use gesture input to select, shrink and zoom in on virtual objects as well as take in gesture volume controls for a music application. COORDINATE SYSTEM

Hand gesture control can be considered as the two sequential problems of hand pose estimation and gesture recognition based on the hand pose predicted. Gesture recognition is most often performed using machine learning approaches such as support vector machines, naïve-bayes classifiers and convolutional neural networks as is done in **Indiansign language** 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-autocorellation and then providing those features to a multi-layer perceprtron or K-nearest neighbors system such as in **combinedsignlanguage** to recognize sign language. The complexity of the algorithm required in gesture recognition depends on the static or dynamic nature of the input gestures as well as the diversity of 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 **handclassicalapproach** relied on classical approaches to hand pose estimation such as using The Continuously Adaptive Mean Shift algorithm to recognize very high or low saturation of image pixels to segment a hand from its background and then using a curvature-based least-square fitting algorithm for detecting the contours of the hand such as fingertips. However, with the advent of modern computing power alluded to earlier and the rise of deep learning, the literature has been saturated with machine learning approaches to hand pose estimation.

A state-of-the-art hand-tracking application created by Google dubbed Mediapipe Hands Google **mediapipe**, uses a series of convolutional neural networks to train a palm detector and hand landmark model on the cropped image of a hand to output hand landmark coordinates. The system runs in real-time on mobile devices and is trained using real images of hands as well as synthetic hand models. Similarly, **deepcnn** uses a deep convolutional neural network with just convolutional and pooling layers to output three-dimensional joint locations for a hand based on webcam input. This is why the literature often refers to hand pose estimation also as hand joint-regression. **handposergbcamera** 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 YOLO9000 **yolo9000**, and then regress the joints of the hand using a large convolutional neural network based on the architecture of RESNet50 **resnet50**.

Alternatively, there are implementations of hand pose estimation that make use of not only standard RGB web camera input but also of depth camera input. This depth input is often modified in an intermediate transform 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 in **poseguidedcnn** where a convolutional neural network regresses joing locations from feature regions which are themselves extracted from feature heatmaps created by depth image input. **cnnfinetuning** and **depthheatmaps** also make use of heatmaps of joint coordinates and finetuning algorithms later to output joint locations based on the intermediate layers and a number of transforms.

handposeocclusions develops an architecture that involves calculating a skeleton-difference loss network to accept depth images as input, segment a hand based on the depth data and intermediate position score map system and then predict hand joints using convolutional layers and a 101-layer recurrent neural network. This implementation is developed to be robust to occlusions of the hand by objects held in the hand as well. Hand pose estimation is a subfield itself of full-body pose estimation which is a problem solved by **deeppose**, which takes advantage of a hierarchial progression of increasingly-fine-grained pose regressors for the joints of a whole body and is comprised at its core layer of simple convolutional layers stacked after each other.

It is evident that the advent of deep learning has yielded a large number of machine learning approaches to not only hand pose estimation but also full-body 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 finding them but rather instead training a machine learning system to identify and learn these low-level abstractions using vast amounts of training data and machine learning architectures such as the convolutional neural network.

In conclusion, to implement a system that can perform hand gesture control of a virtual object in augmented reality it is evident that the preferred approach in the literature for such a task is to use a deep-learning approach to regress hand joint coordinates from either a depth or standard RGB camera. Gesture recognition can either be performed by machine 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 to capture video and depth data about the environment. A system will thus be developed that can accept user gesture input using a deep-learning approach and then translate that gesture into meaningful instructions for a virtual object present in an augmented reality scene that is tied together using a global coordinate system and presents realistic physics and interactions between the objects based on a simulated collision avoidance system.

OpenGL Movement API

The objective of this work session is to create a small API for converting normalized xyz world coordinates into OpenGL coordinates so that the virtual object and real world pixel coordinates can be compared. This is accomplished by setting an original 0,0,0 coordinate for the virtual object when the program is initialized and updating it based on the passed in coordinates and the glTranslatef OpenGL command. Since the display window

is set to be 800x600px the glTranslate method can take in a maximum of 8.0 and 6.0 for the x and y parameters respectively. Pictured in Fig. **[6.1]** is the cube at world coordinate (0,0) and the result of moving it there with gesture input provided by the mediapipe library.

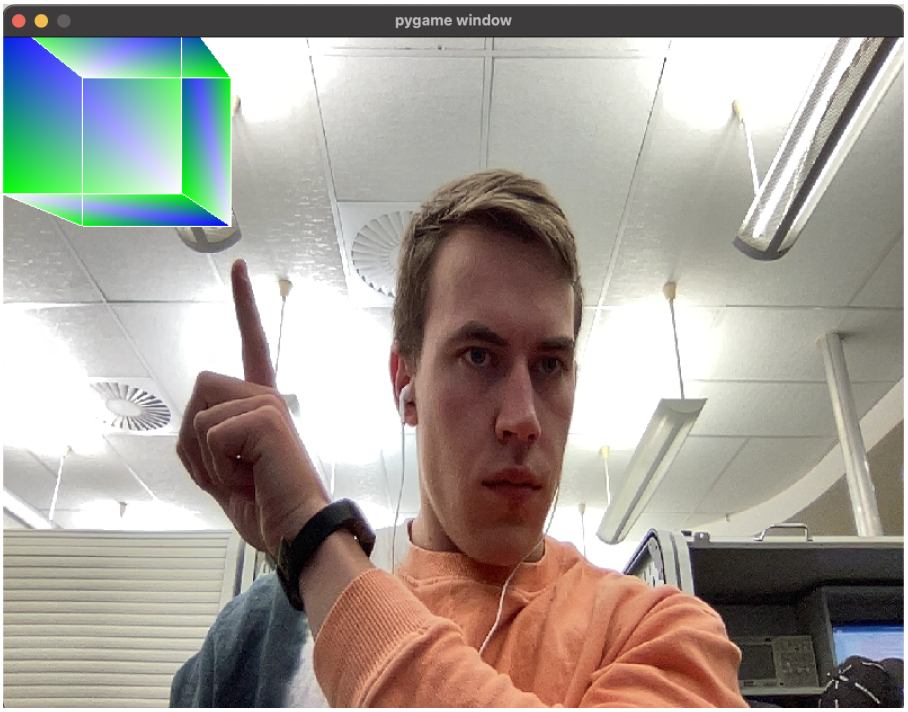


Figure 6.1: OpenGL virtual object at world coordinates 0,0.

Tuesday, 02 August 2022

Neural network convolutional layers

The aim of the work session today is to get multiple convolutional layers working with each other in the layers version of the convolutional neural network. This involved refactoring code so that output from a previous convolutional layer could be accepted as input to the next one - this mainly consisted of changing the order of for loops and iterations through the dimensions of the filter arrays and outputs so that multiple filters with x and y components with multiple channels $filters[numFilters][x][y][numChannels]$ could match with other variables like the convolutional loss gradient from following layers $convolution\_gradient[numFilters][x][y][numChannels]$.

Wednesday, 03 August 2022

More training data for gesture classifier

The aim of today's work session is to build up a large dataset of hand coordinates and their corresponding images for better training.

The dataset can be found in the repo in the "handcoordstrainingdata" folder. It was attempted to train the network on this data but it took over an hour for just 5 epochs. Attempts were made to shrink the input images to dimensions of just 20x15 but at this resolution not enough data remained for good generalization. Preprocessing was introduced in the form of the edge detection filter and convolution previously used in experimentation to detect the edges of objects using the Kinect sensor. This made the data much simpler and stripped away a lot of unnecessary background information and as of this writing training is still ongoing. It is hypothesized that in the final implementation of the system, it is going to be essential to perform preprocessing in this manner if the network is going to run in any efficient timeframe. Additionally, developing a hand segmentation algorithm or hand location algorithm would also be highly helpful and in-line with the industry-leading approach Mediapipe which does this to ensure that most of the learning capability of the network goes towards recognising a hand and not dealing with background information.

The industry standard activation function is the Relu function but using it in the place of the sigmoid function currently used gives garbage output from the CNN and a static epoch error rate. Perhaps the outputs of various layers need to be normalized or simplified to use this function but getting this working is a priority as it brings the architecture of the system in-line with known working solutions and is estimated to dramatically increase training and inferrence time as certain neurons and weights are saturated when they do not provide helpful information to the network. Additionally, implementing dropout regularization is a future goal as well.

Thursday, 04 August 2022

Kinect Segmenting Prototyping And Keras Gesture learning

The aim of the work session today is to experiment with using the depth values from the Kinect sensor for segmenting images and to use Keras to attempt to build a small gesture classifier using the same training data provided to the layers neural network developed from first principles.

The results of the image segmentation based on the Kinect depth data can be seen in Fig. **[6.2]** and Fig. **[6.3]**. The Kinect returns depth values from 0-2048 with values closer to the sensor being labelled closer to 0. With a threshold in place here to only display the pixel values where the corresponding depth value is less than 150 a segmentation algorithm can be created that only displays those things close to the sensor. This is potentially valuable in removing excessive background noise or information from processing. Additionally, it confirms that the Kinect sensor can only determine depth values from about 50cm away as any closer yields noise values.

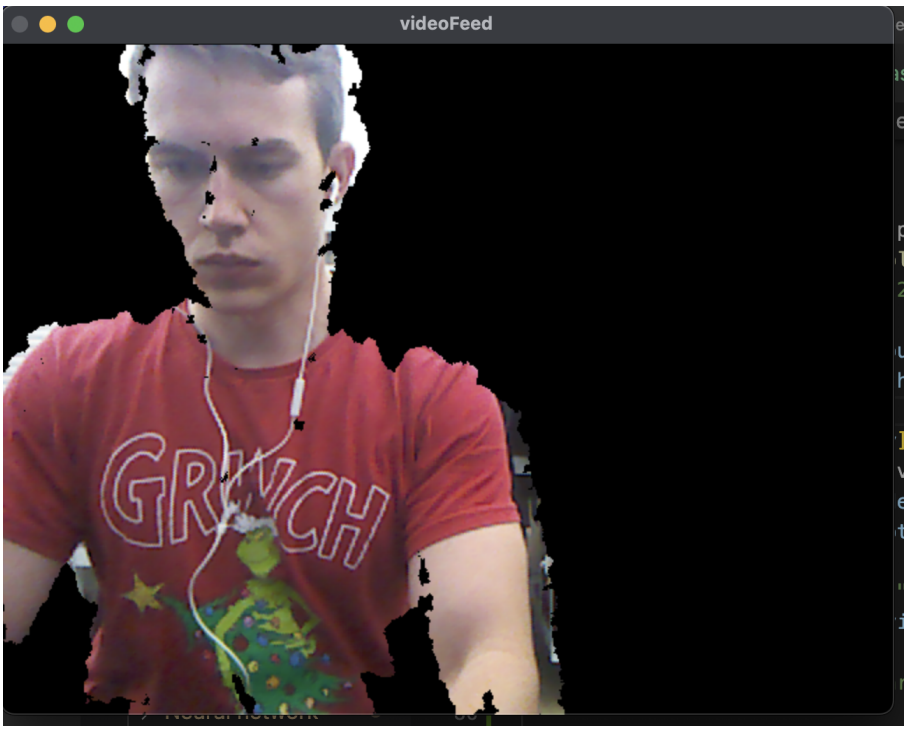


Figure 6.2: Segmenting an image based on the Kinect depth data

TeachableMachine and Keras was used to build a small fist/open hand classifier and the model downloaded and examined - only sequential or "hidden" layers were used and could accurately classify the data found in gesturetrainingdata. Removing the convolutional layers from the network also allowed the first principles implementation to classify these images correctly as visible in Fig. **[6.4]** and Fig. **[6.4]**. The network is definitely overfitted on the training data but it is a good first step in recognising gestures using webcam input.

Thus it is hypothesized that something is wrong with the convolutional layers and mathematics of the first principles implementation as a basic image classification task can be accomplished using just hidden and output layers.

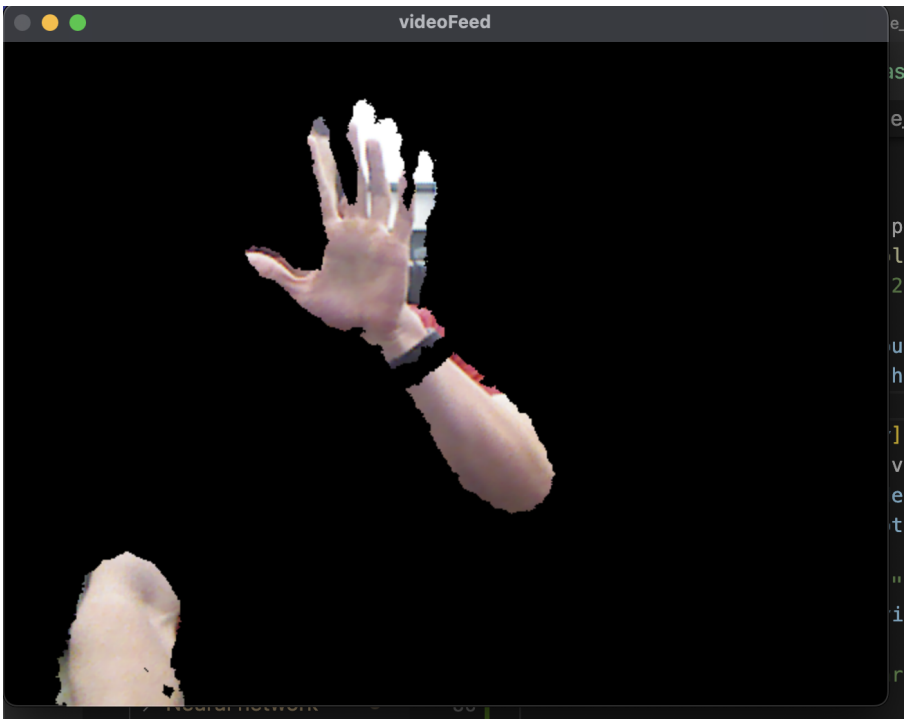


Figure 6.3: Segmenting an image based on the Kinect depth data

Figure 6.4: First principles open/closed hand predictor

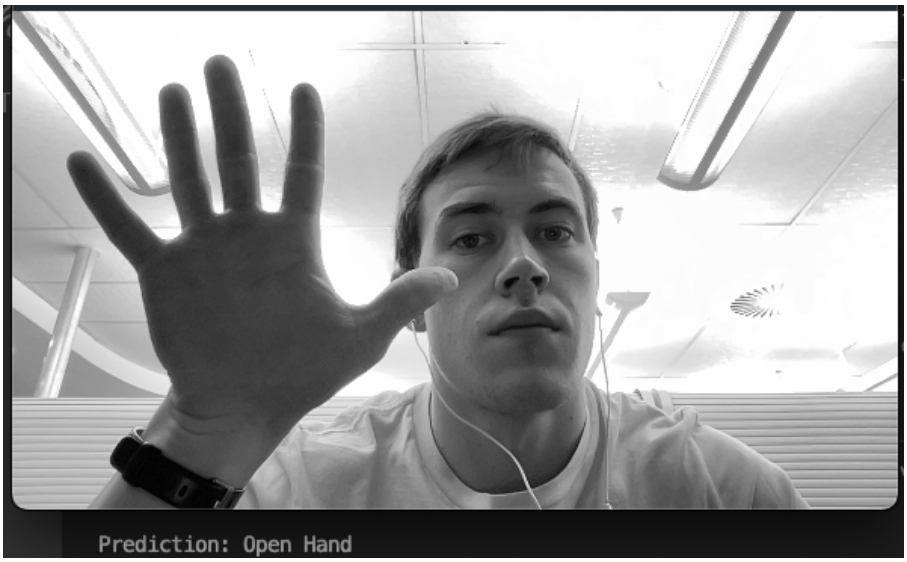


Figure 6.5: First principles open/closed hand predictor

Friday, 05 August 2022

First Semester Report Submitted

The first semester report was submitted and the final version of the literature review - with correct referencing, is including below.

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.

These two sub-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 Billinghurst [10] who utilizes a Microsoft Kinect depth and RGB camera to treat agoraphobia by creating virtual spiders that the user can interact with using their hands in an augmented reality application. Baldauf [11] 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 recognize gesture volume controls for a music application.

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 [12] implements a tabletop augmented reality application for handling small virtual shapes 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. Similarly, Buchmann [13] 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.

These applications receive hand gestures as input and hand gesture control itself can be considered as the two sequential problems of hand pose estimation and gesture recognition based on the hand pose predicted. Gesture recognition is most often performed using machine learning approaches such as support vector machines, Naïve-Bayes classifiers and convolutional neural networks as by Ahmed [14] 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 [15] to recognize 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 [16] relied on classical approaches to hand pose estimation such as using The Continuously Adaptive Mean Shift algorithm to recognize very high or low saturation of image pixels to segment a hand from its background and then using a curvature-based least-square fitting algorithm for detecting the contours of the hand such as fingertips. An alternative to hand pose estimation is to use a physical glove with fiducial markers on it as used by Buchmann [13] 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 special hardware or highly specific algorithms that previous implementations required.

A state-of-the-art hand-tracking application created by Google - dubbed Mediapipe Hands [8], uses a series of convolutional neural networks to train a palm detector and hand landmark model to output coordinates of hand landmarks. The system runs in real-time on mobile devices and is trained using real images of hands as well as synthetic hand models. Similarly, Qing [17] uses a deep convolutional neural network with just convolutional and pooling layers to output three-dimensional joint locations for a hand based on webcam input. This is why the literature often refers to hand pose estimation as hand joint-regression. Gomez-Donoso [18] 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 [19], and then regress the joints of the hand using a large convolutional neural network based on the RESNET50 architecture [20].

Alternatively, there are implementations of hand pose estimation that also make 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 [21] where a convolutional neural network regresses joint locations from feature regions which are themselves extracted from feature heatmaps created by depth image input. Ding [22] and Ge [23] also make use of heatmaps of joint coordinates and subsequent fine-tuning algorithms to output joint locations based on the intermediate layers.

It is evident that the advent of deep learning has yielded a large number of machine learning 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 machine learning system to identify and learn these low-level abstractions using vast amounts of training data and machine learning architectures such as the convolutional neural network.

In conclusion, to implement a system that can perform hand gesture control of a virtual object in augmented reality it is evident that the preferred approach in the literature for such a task 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 machine 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. A system will thus be developed that can accept user gesture input using a deep-learning approach and then translate that gesture into meaningful instructions for a virtual object present in an augmented reality scene that presents realistic interactions between the objects and uses a global coordinate system to prevent object collisions.

Fixing Dying Relu Problem

The use of the relu function as the activation function has long given errors in the first principles implementation as after a single epoch of training the error remained constant and no learning could take place. This was hypothesized to be because of a dying relu problem where the output of a relu function is almost 0. The output of each layer and the current value of the hidden and output weights were printed to the network details

textfile and upon examination it seemed that the weights were initially set and then never really changed again. Changing the weight initialization to much smaller values, like -0.1 to 0.1, seemed to alleviate this problem but the network shortly converged to the same epoch error and stayed there. Research was done into the effects of weight initialization, batch normalization and regularization and these operations may have to be implemented in order to get the relu activation function working correctly.

Retraining of basic gesture classifier with convolutional network

It was attempted to train the network using one convolutional layer and one hidden layer to predict the open/closed hand gestures. However, after 15 hours of training the learning was still ongoing on a 2020 Macbook Air M1 laptop. Alternative training approaches have to be considered if the project is to be completed on-time - at this rate of training it is infeasible to test different designs. The use of Google Colab notebooks was briefly investigated however since the first principles implementation doesn't use a GPU or TPU-optimized framework like Tensorflow or Pytorch the advantages of a GPU or TPU are wasted and the implementation actually ran slower than on the Macbook Air PC.

Friday, 05 August 2022

Brainstorming collision avoidance pipeline

The training for the CNN to classify open/closed hand gestures is still ongoing.

In order to create realistic interactions between the virtual object and real-world objects it is going to be necessary to perform collision avoidance when attempting to move the virtual object. A proposed pipeline for this process is presented in Fig. **[6.6]**.

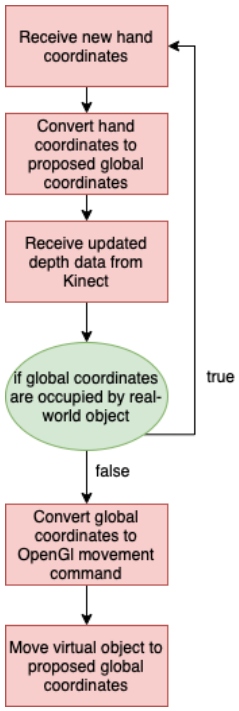


Figure 6.6: Proposed Collision Avoidance Algorithm.png

The training of the hand closed/open network was terminated after 20 hours and still an epoch error of over 40. Additional modifications to the network and training process are obviously necessary to achieve the desired performance. Specifically, segmenting the hand from the background is going to be a necessary step to remove complexity from the neural network and focus on classifying hand coordinates/gestures and not whole body shapes. This is what [8] did. Thus in the aim of building a hand detector the Ycbcr color space is used to experimentally interpret the values from a webcam, since skin hue color is a specific color and can be segmented from other colors - a computationally cheap way of identifying skin in an image. The Ycbcr color space is defined as " Y the luma component and CB and CR are the blue-difference and red-difference chroma components." A webcam image converted into this color space is presented in Fig. **[6.7]** and thresholding using the values of the color space is presented in Fig. **[6.8]**. Background colours similar to the colour of the hand provide background noise however it clearly segments the hand and face based on skin colour and is a promising approach for hand segmentation.

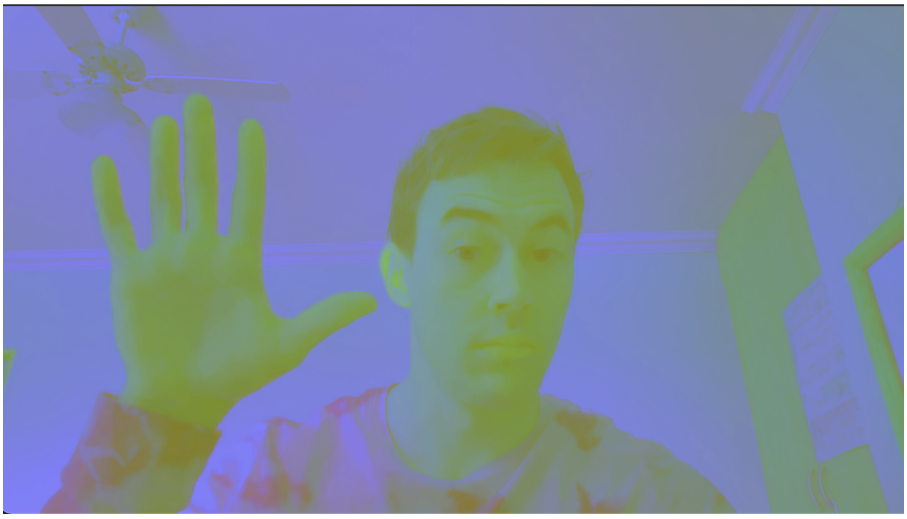


Figure 6.7: Ycbcr-encoded webcam image

Figure 6.8: Ycbcr segmentation

Tuesday, 09 August 2022

Brainstorming segmentation and more efficient algorithms

Time was spent during this work session researching the Ycbcr color space further and experimenting with it. Several research papers combine color space and depth information in order to correctly segment hands from an image. The computational speed of searching through the entire depth and color arrays provided as input from cameras is very inefficient - using built-in Python functions such as array indexing with code like $result = np.where((array > 2), array, 0)$ to perform thresholding on values smaller than 2 can drastically reduce runtime and improve performance. Produced in Fig. **[6.9]** is the handwritten lab notes entry from today brainstorming algorithm processes and the thresholding pipeline for hand detection.

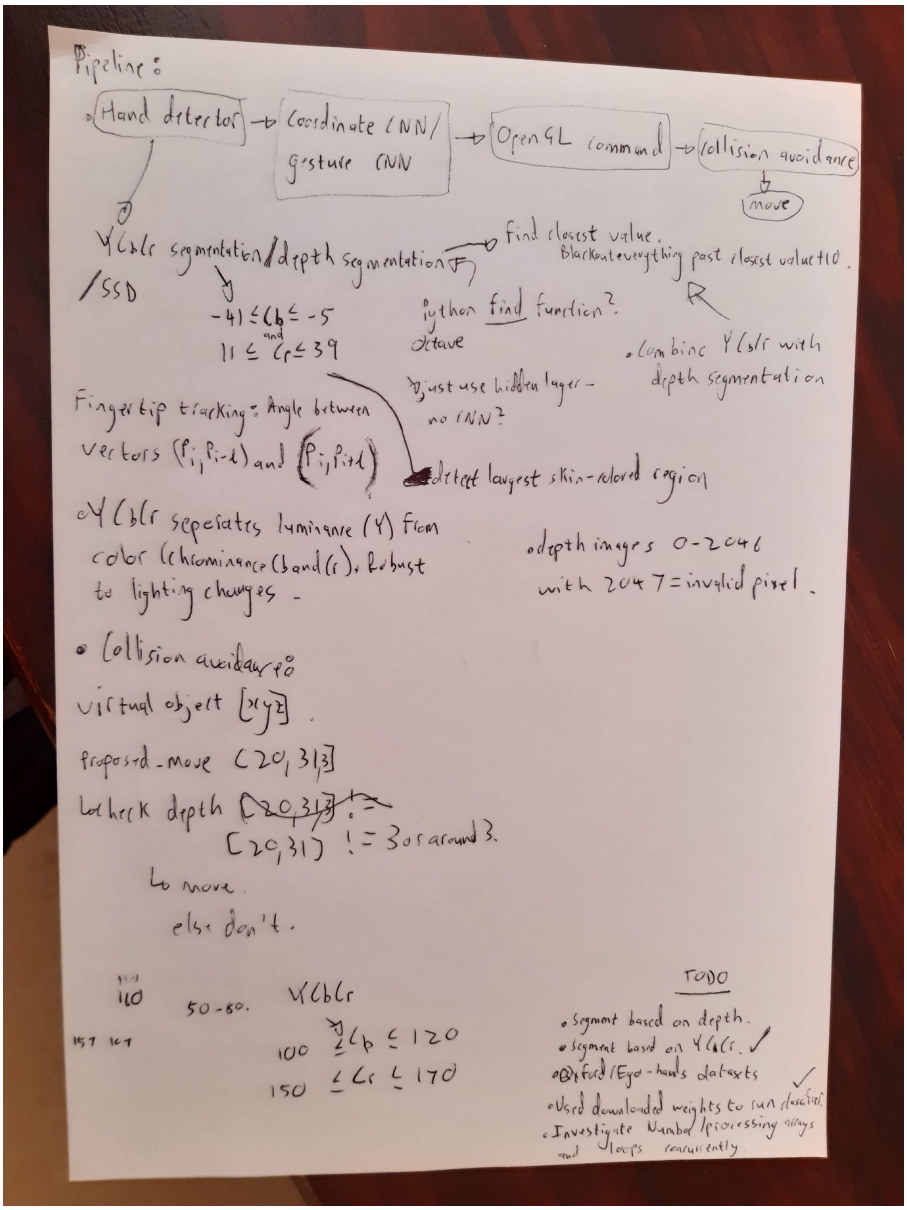


Figure 6.9: Handwritten notes brainstorming color and depth segmentation

Thursday, 11 August 2022

Using the Keras and Tensorflow libraries to train networks

The aim of this work session is to use Keras and Tensorflow to build a gesture recognition system.

In a meeting with Mr. Grobler on the Wednesday, 10 August 2022, the excessive slowness of training neural networks using the first principles implementation reported on earlier in this document was discussed. It was decided that it would be prudent to use an optimized library like Tensorflow to conduct the training of the gesture classifer neural network in order to speed up the process to tolerable wait times. This was agreed too not to be in violation of constructing all systems of the Project from first principles due to the actual implemented system (the gesture classifier and network) will still work with a first-principles forward-propagation implementation and no libraries. However using a library to obtain the correct weights of the network was deemed a fair use of existing tools since the goal of the project is not to build an end-to-end first principles machine learning pipeline but rather achieve the objective of real-time gesture recognition. Additionally, the first-principles implementation of the training and backpropagation process can be shown to work albeit slowly and discussed and compared to existing libraries in the final report.

Thus learning commenced yesterday on August 9, 2022 and experimentation with the Keras and Tensorflow libraries began. This resulted in the construction of a reliable open-hand/fist classifier using the architecture shown in Fig. **[6.10]** with good accuracy as shown in Fig. **[6.11]**. The output of the system on live webcam input is shown in Fig. **[6.12]** and Fig. **[6.13]**.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 116, 156, 32)      320

 max_pooling2d (MaxPooling2D  (None, 58, 78, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 56, 76, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 28, 38, 64)       0
 2D)

 flatten (Flatten)           (None, 68096)             0

 dense (Dense)               (None, 2000)              136194000

 dense_1 (Dense)             (None, 2)                 4002

=================================================================
Total params: 136,216,818
Trainable params: 136,216,818
Non-trainable params: 0
```

Figure 6.10: Architecture of Keras Open Hand/Fist Classifier

Much of the day was spent attempting to extend the result of the open hand/fist classifier to multiple gestures, however although an accuracy of 100% could be attained on the training data, only about 80% could be achieved on the testing data. This shows possible overfitting to the training data and a lack of training data overall in order to get the network to generalize the hand gesture shapes. Data augmentation was performed by flipping all the data across the vertical axis and essentially doubling the amount of training data. At the time of writing, the training process using this data was still underway.

```
198/198 [==============================] - 47s 234ms/step - loss: 2.4191 - accuracy: 0.5152 - val
_loss: 0.6655 - val_accuracy: 0.6818
Epoch 2/3
198/198 [==============================] - 47s 239ms/step - loss: 0.4755 - accuracy: 0.7677 - val
_loss: 0.3677 - val_accuracy: 0.9091
Epoch 3/3
198/198 [==============================] - 51s 257ms/step - loss: 0.1001 - accuracy: 0.9646 - val
_loss: 0.0943 - val_accuracy: 1.0000
```

Figure 6.11: Accuracy and loss of Keras Open Hand/Fist Classifier

Figure 6.12: Prediction of Keras Open Hand/Fist Classifier

What is of note is that the input to the network after edge detection currently is as seen in Fig. **[6.14]**. A lot of background information is still retained in this image and may be confusing the network or allocating learning capability to classifying the background when it is useless. To this end, further segmentation was performed using the HSV and Ycbcr color spaces with masking this time around in order to more efficiently segment skin from background imagery. The results are visible in Fig. **[6.15]** and Fig. **[6.16]** and it is clear that the Ycbcr color space segmentation is still superior.

This segmenting will thus be integrated into the preprocessing stage of the network and is hypothesized to increase training accuracy and the system's performance.

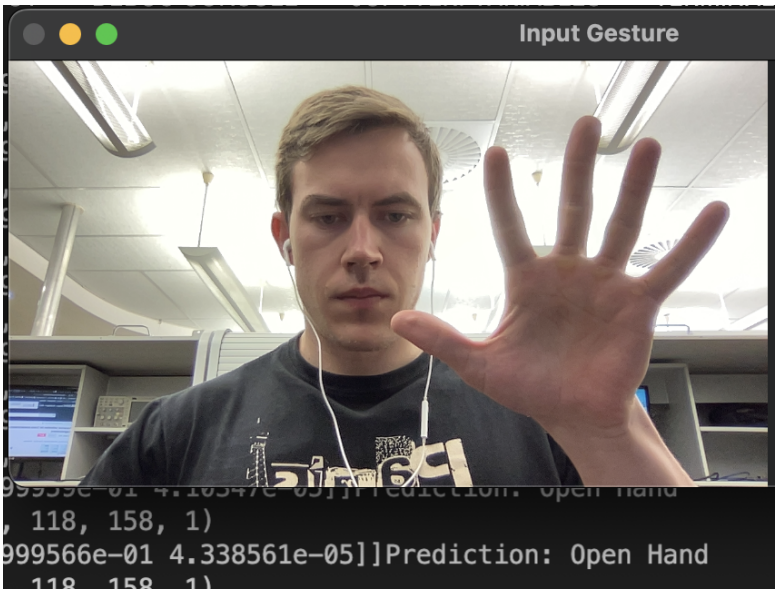


Figure 6.13: Prediction of Keras Open Hand/Fist Classifier

Figure 6.14: Current input to network after segmentation

Figure 6.15: Segmentation using masking and the HSV color space

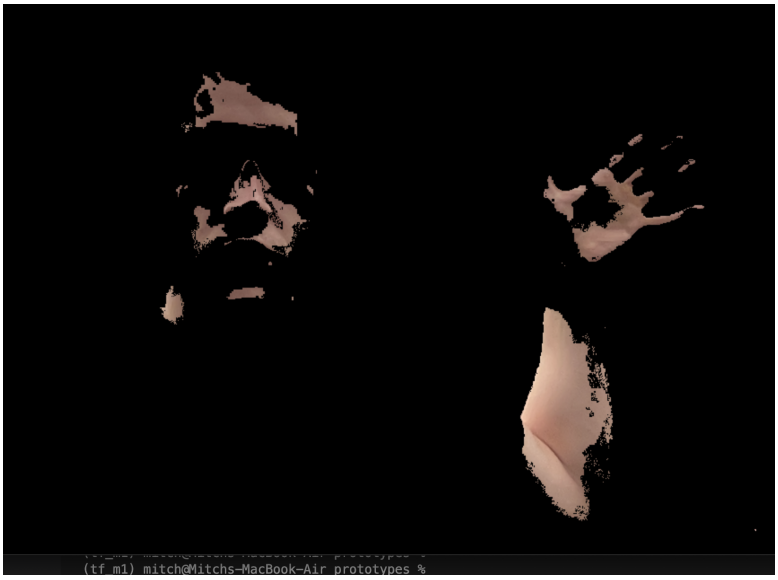


Figure 6.16: Segmentation using masking and the Ycbcr color space

Monday, 15 August 2022

Discussion of current gesture recognition system inadequacies

The use of Keras to train and build prototype gesture recognition systems has met with success. As shown above, these prototypes can tell the difference between an open and closed hand with good accuracy on both a training and testing dataset however real-world performance suffers when the system is provided with novel input from a webcam. The system is not yet good enough to accurately classify gestures in new and novel positions. A number of optimizations can thus be pursued to increase the accuracy of the system. These are increasing the amount of training data, increasing the size of the network and implementing a hand detector system to reduce the amount of input given to the system.

On this last suggestion, the current input to the system is the segmented skin region parts of an image as seen in Fig. **[6.17]** and although this considerably simplifies what the system has to learn, it still contains the parts of the image corresponding to the user's face and it is hypothesized that this extra input is confusing the network or rather wasting its resources trying to classify parts of the face that aren't relative to the hand gesture being displayed. Thus the construction of a hand detector that can recognize the pixels of the image that belong to the hand and segment only them from the background could be a way to reduce the input complexity to the system and improve gesture detection accuracy.

Training the network with input data that only contains images of hands in various gestures without faces present in the images is also a possible way of increasing the accuracy of the system and will be attempted as well. Thus the next tasks to complete in order to increase the accuracy of the gesture recognition system are as follows.

- Re-train gesture recognition network using input images without faces present
- Increase amount of training data provided to system
- Increase size of the network used for classification
- Construct a hand detector to further segment the hand from the remainder of the input image

Figure 6.17: Current segmented input to the system

While attempting to re-train the network with images devoid of faces it was discovered that the system is sensitive to lighting changes and the edge detection filter outputs close-to-junk data if the lighting conditions change drastically. The lighting changes between two sets of input data are shown in Fig. **[6.18]** and the junk data result of the edge detection filter with a different lighting condition to what it was initially setup with is shown in Fig. **[6.19]**. Thus for further experimentation, sensitivity to lighting conditions will have to be taken into account

in order to achieve consistent output. The importance of cleaning data before it is fed into a machine learning system is thus clearly demonstrated by this problem and will be rectified in future.

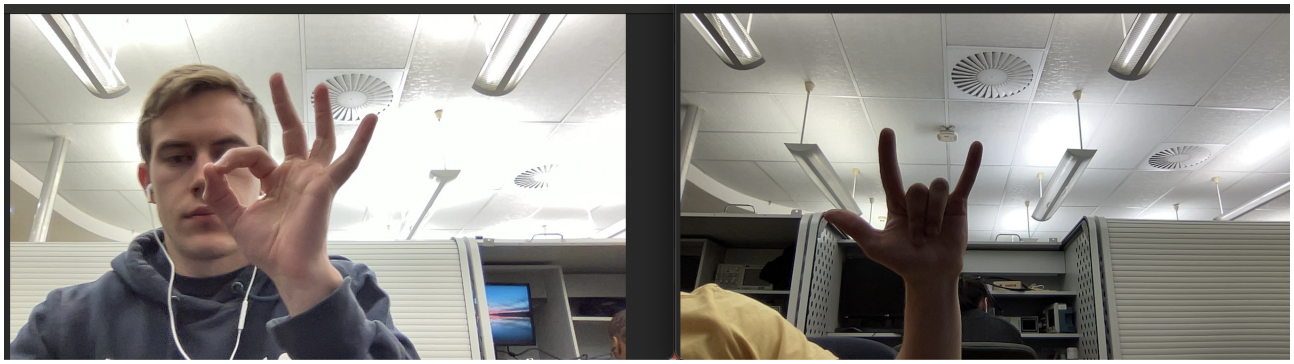


Figure 6.18: Different lighting inputs to the gesture recognition system

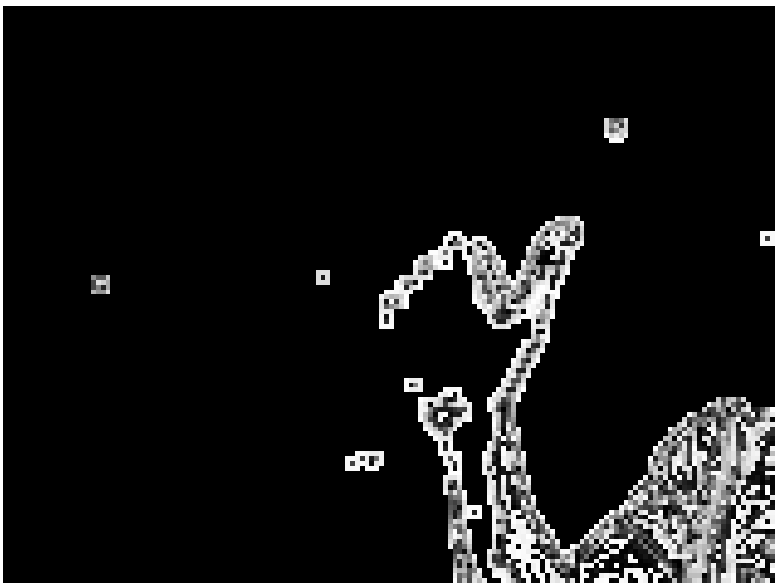


Figure 6.19: Current segmented input to the system

Tuesday, 16 August 2022

Construction of updated prototype implementation

The aim of today's work session is to construct up-to-date prototypes of the system.

Previously, an OpenGL movement API was developed that could take in a value between 0 and 1 for each an x, y and z coordinate value and normalize this to the width and height of the screen. This normalized value would then be the destination coordinates for a cube rendered in OpenGL. Because OpenGL only has rotations and translations a small API was developed that could take in these world coordinates and transform them to the corresponding translation and rotation commands that OpenGL could render and appear to move the cube to the correct coordinates. This was done by finding the delta between the current and desired coordinates and translating the OpenGL cube by these delta amounts as well as adding extra translations in the x and y directions when the cube was translated in a z-direction in order to keep it in place but correctly "grow" or "shrink" the cube from the user's perspective. This movement API was used extensively in the following prototypes.

The first prototype is built using OpenGL and Mediapipe and it is a pinky-tracking cube movement system. The location of the pinky finger in a webcam input is found using Mediapipe and the OpenGL cube movement API designed from first principles accepts a coordinate as input and then snaps the cube to the location of the pinky finger. In this way, rudimentary tracking can be achieved. The result of the prototype can be seen in Fig. [6.20].

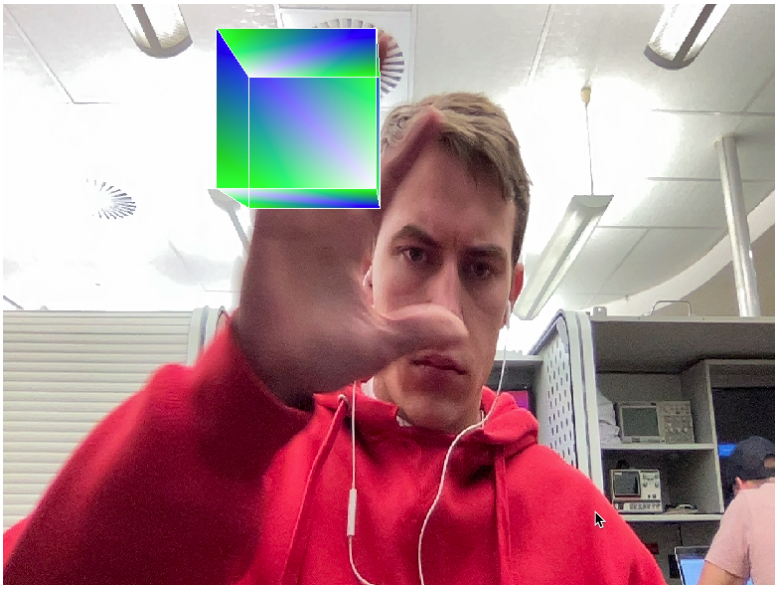


Figure 6.20: OpenGL and Mediapipe Pinky Tracker Prototype

The second prototype is built using the same OpenGL movement API developed but this time with a small Tensorflow convolutional neural network for detecting whether input webcam images contain a closed fist or open hand. The operation of this system is visible in Fig. [6.21]. Based on the detected gesture the cube is either snapped to the left or right of the screen. The accuracy of the model is 1.0 on the training data and nearly the same on the test dataset however it performs with middling accuracy on real-live webcam input. This is once again hypothesized to be due to the changing lighting conditions when a hand is adjusted slightly so that it reflects more of the ceiling lights - this is affecting the segmentation algorithm which discriminates a hand based on skin colour and illuminance and so optimizing the network to be robust to changing lighting conditions is necessary in order to get it to accurately recognize gestures.

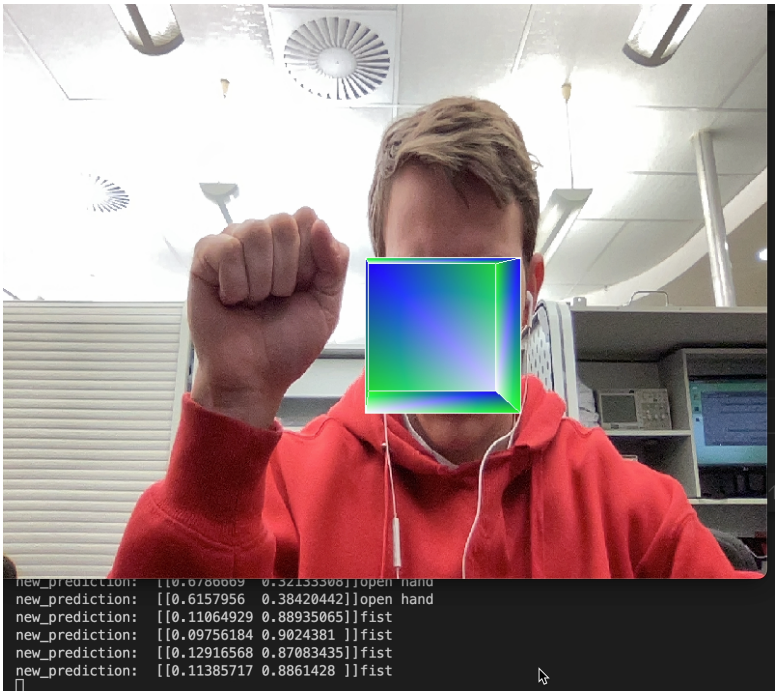


Figure 6.21: OpenGL and Tensorflow Fist/Open Hand Prototype

The final prototype constructed uses the same OpenGL movement API and the Kinect sensor to perform object collision avoidance between the virtual object and real-world environment. The x and z keys are pressed on the keyboard and alternatively move the cube to two different locations on the screen - which have corresponding x and y coordinates. At the same time, the Kinect sensor receives depth data for the entire width and height of the screen and this depth data is examined for a certain value in a range between 500 and 700, which is roughly a metre and a half from the camera. If there is a value in this range anywhere near the x and y coordinates that the cube is about to be moved to, the cube is prevented from moving there and an object "collision" is registered. The operation of the system is visible in Fig. **[6.22]** and shows the user trying to move the cube back to the top left of the screen but the system preventing the action from taking place because the user is located there and is taking up the space there. This basic prototype shows that the Kinect depth data can be used to prevent object collisions between real and virtual objects with moderate speed and accuracy. Scaling up the object detection system and matching the OpenGL cube coordinates to those of the image is the next major hurdle.

Additionally, after attempting many methods, the exposure of the webcam used to generate training images and perform hand gesture inferences was set to a constant value using the Camera Controller application. The settings are visible in Fig. **[6.23]** and are hypothesized to make training the hand gesture recognition system much more reliable as segmentation based on the Ycbcr colorspace will be consistent and not changed by arbitrary lighting conditions.

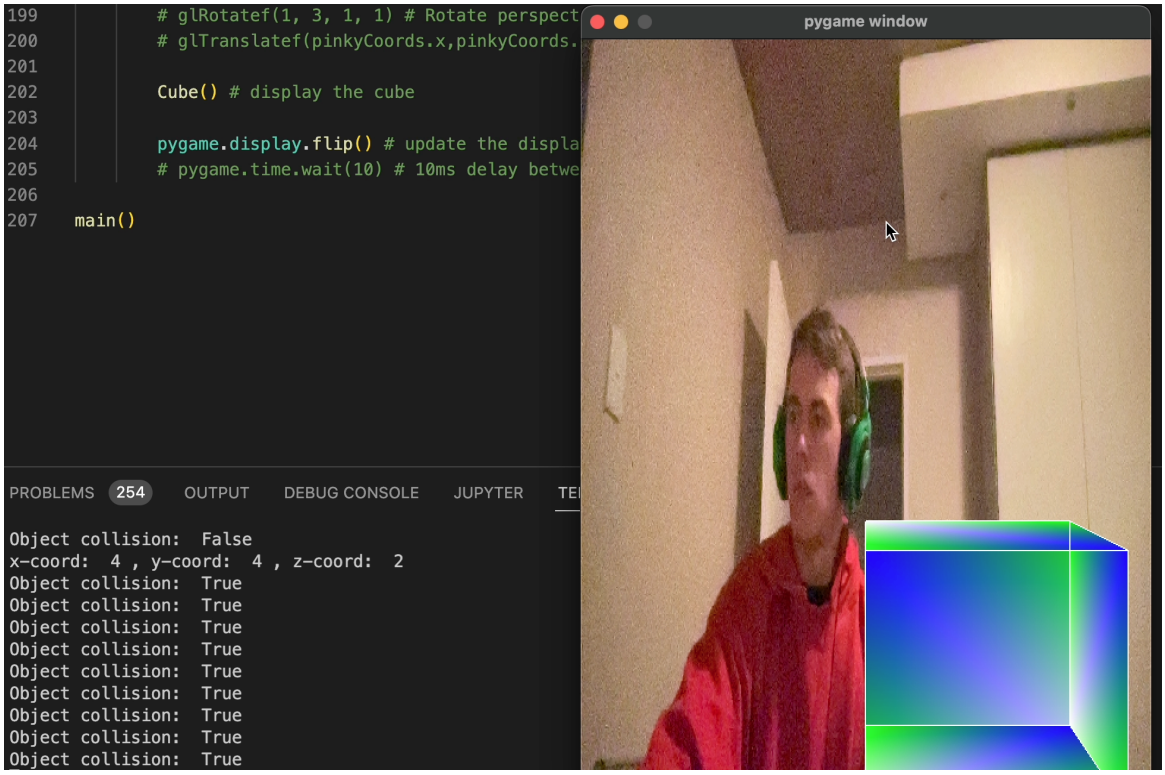


Figure 6.22: OpenGL and Kinect Collision Avoidance Prototype

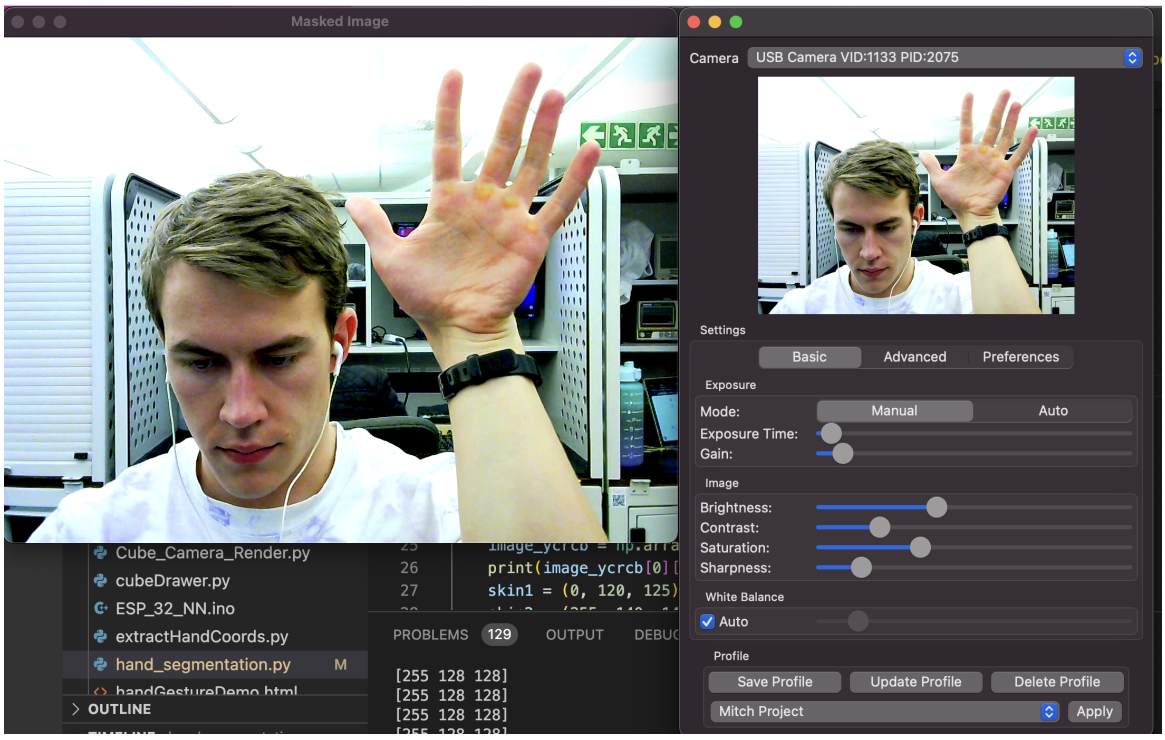


Figure 6.23: Camera controller settings

Thursday, 18 August 2022

Gesture recognition system changes and demonstration planning

The final objective of the system is to perform real-time gesture control of a virtual object in augmented reality. This will be demonstrated in a physical demonstration and thus the placement of the various cameras will affect the final performance of the system and how the training and testing will take place. The proposed hardware setup is illustrated in Fig. **[6.24]** and shows how the Kinect camera will be placed to the side of the table and the the webcam positioned above the desk. This is so that the virtual object can be interacted with on the tabletop and the depth data for the table and distance from the camera calculated. Furthermore, it simplifies the hand recognition/segmentation by using a top-down view which will not have the user's face in frame but rather just their hand and part of their arm. This gave rise to potential simplifer solutions for identifying the location of the hand in the frame such as simple color segmenting and even a contour-based approach, which will speed up the development and implementation considerably. This top-down view is presented in Fig. **[6.25]**

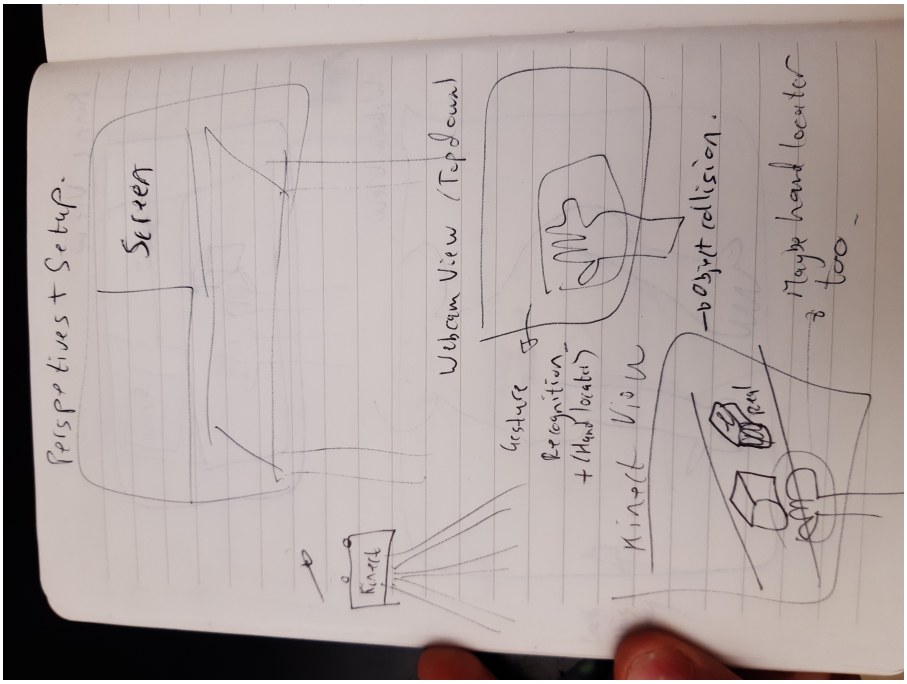


Figure 6.24: Proposed camera and hardware setup for demonstration

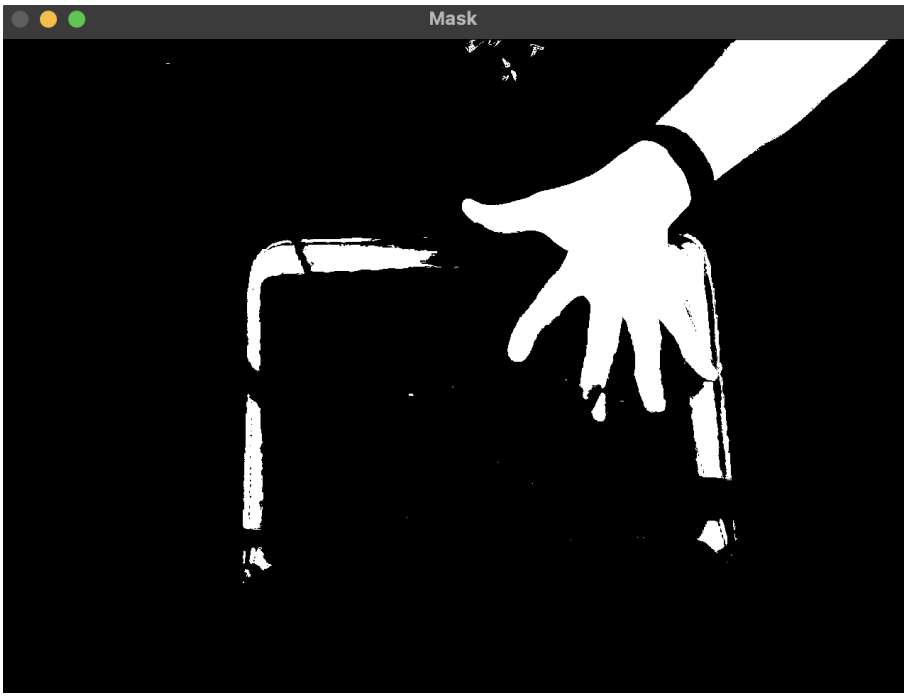


Figure 6.25: Proposed camera top-down view

Friday, 19 August 2022

Hand locator

The aim of this session is to build a hand detector system.

In order to do this a CNN will be used and trained on a collection of top-down pictures of hands present in the image. A tool for labelling these hands has been built using OpenCV and the images can be labelled by moving the rectangle over the hand using the keyboard and then saving the position of the box. The image is split into 22x17 = 374 individual regions which can either have a hand present in it or not. One of these regions is shown overlain on the hand in Fig. **[6.26]**.

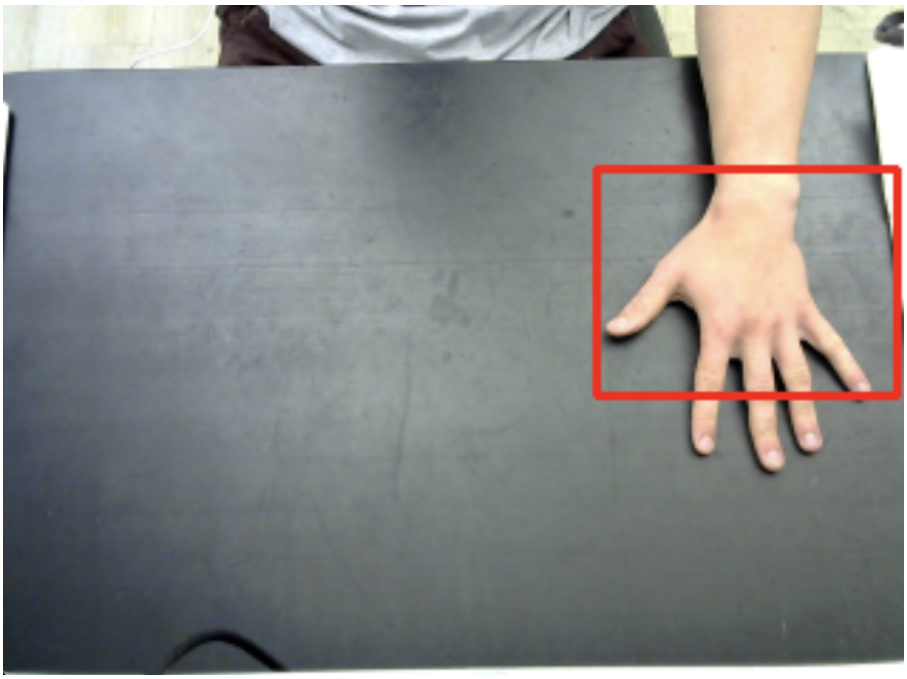


Figure 6.26: Hand locator tool

For all of the training images, the centre of the palm was determined by the user placing a rectangle over the hand in an interactive program and then extracting the image width and height coordinate from the centre of that rectangle. Delta values were then computed for all the candidate regions in the image based on their distance from the extracted palm coordinate. This is visible in the heatmap shown in Fig. **[6.27]**.

After much experimentation it was realised that the system architected above with the heatmap and delta values was incapable of being learned by a convolutional neural network in its current state. There is just too much complexity and not enough training data for the network to generalize that a specific point on the hand was representative of the 0 delta value and training just saw exploding loss values.

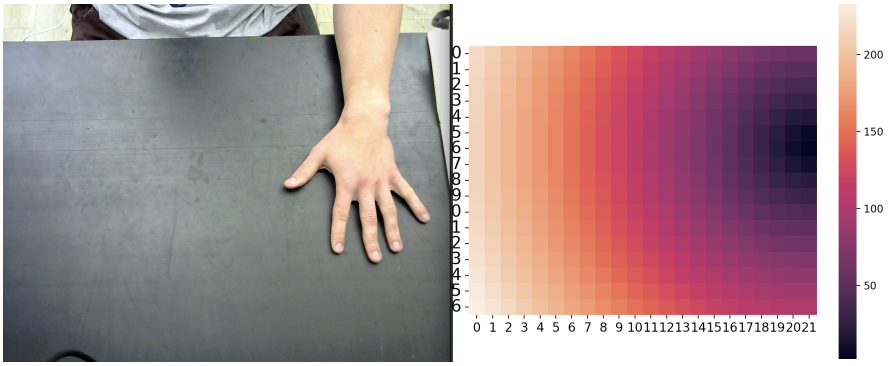


Figure 6.27: Heatmap of hand detector

Tuesday, 23 August 2022

Hand locator changes

The aim of this session is to continue work on the hand locator and construct up-to-date prototypes using as many first principles components as possible.

Because the heatmap delta neural network architecture is not working as desired, a new method of detecting the hand was devised - using classical methods. The mask of the Ycbcr-segmented image clearly shows a concentration of white pixels where the hand is located. By creating a number of candidate regions split across the image and looping through each region and summing the pixel values there, the general location of the hand can be identified. This is visible in the prototype constructed in Fig. **[6.28]**. The system is mostly accurate except for when other large patches of skin are visible in the frame and when the hand in the frame is contorted in different manners.

Since the camera is in a top-down orientation, the user's arm will always enter from the top of the video frame. Thus finding the largest region of skin-coloured pixels nearest to the bottom of the screen will effectively account for error and will not detect the large part of the forearm instead of the hand. This was implemented by calculating a south delta value for each region which corresponds to its distance from the bottom of the screen. Finding the smallest value in this list gives the southern-most region of skin pixels in the frame. Finding the largest region of skin pixels within a certain range of this southern-most point yields the hand or palm region consistently. Thus the hand tracker is complete and will be left as is for now.

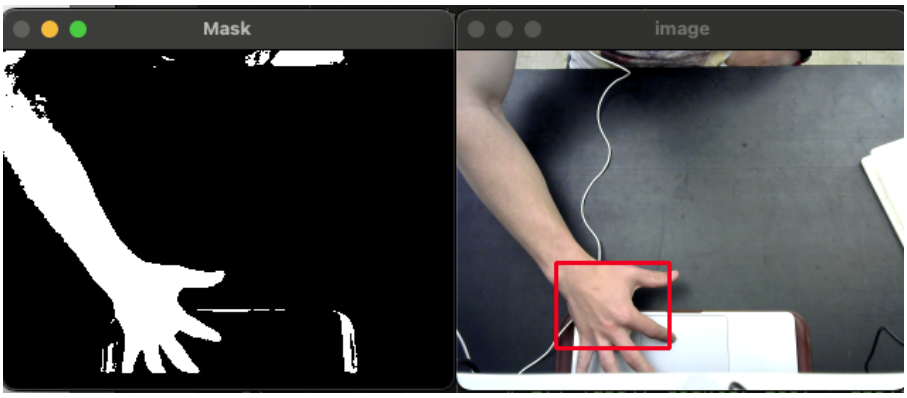


Figure 6.28: Hand detector using mask concentration algorithm

Wednesday, 24 August 2022

World Coordinate System

The aim of this session is to plan and prototype the world coordinate system.//

In order to have meaningful interactions between virtual objects and real objects a shared coordinate system must be established. This can be in the form of an x and y coordinate - being the width and height values of the input image as well as a depth value. The Kinect sensor outputs values between 0 and 2047 representing how far away a specific pixel is from the camera and thus giving virtual objects this coordinate too and growing/shrinking them according to this value will make sense. The planning for said system is visible in Fig. **[6.29]**.

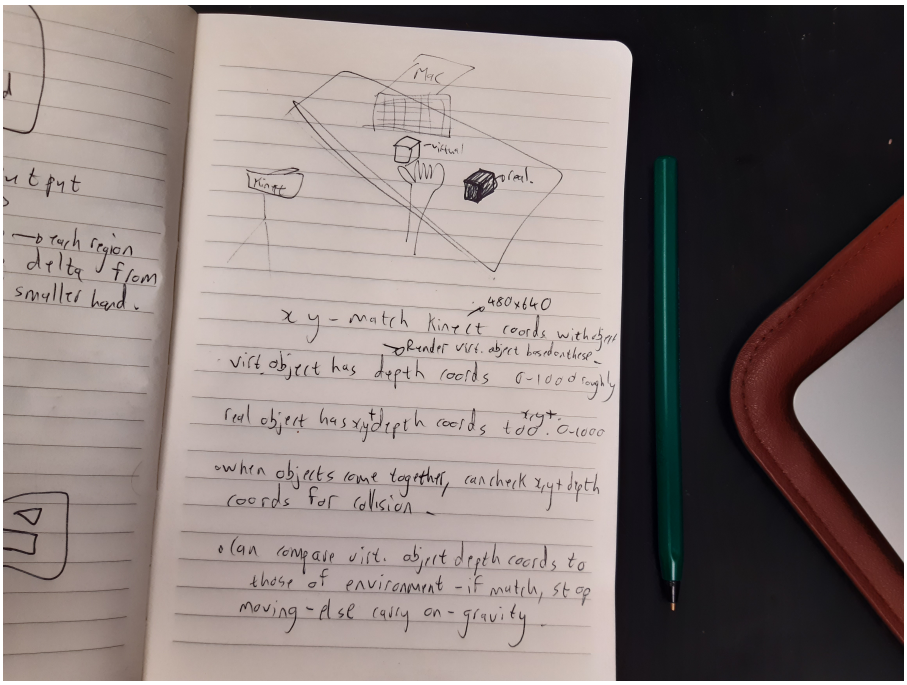


Figure 6.29: Planning of the Kinect and virtual object coordinate system

Thursday, 25 August 2022

Virtual Object Rendering Changes

Using OpenGL was initially decided upon because it was the industry standard for 3D graphical applications. However, interfacing the projection-based three-dimensional coordinates system with the two dimensional xy coordinate system used by the Kinect and ordinary webcams has proved challenging. In order to aid further development of the virtual object prototyping system, a prototype was constructed for rendering a virtual cube using OpenCV - which shares the same two-dimensional xy coordinate system as the Kinect. Building this prototype requires manually specifying each vertex of the cube in two dimensions and then rendering three distinct polygons which are the three visible faces of the cube. This is demonstrated in Fig. **[6.30]**. Further prototyping will be needed to enable the cube to be rotated and generalized to any point on the xy coordinate system but it has removed a lot of transitional overhead for translating between coordinate systems.

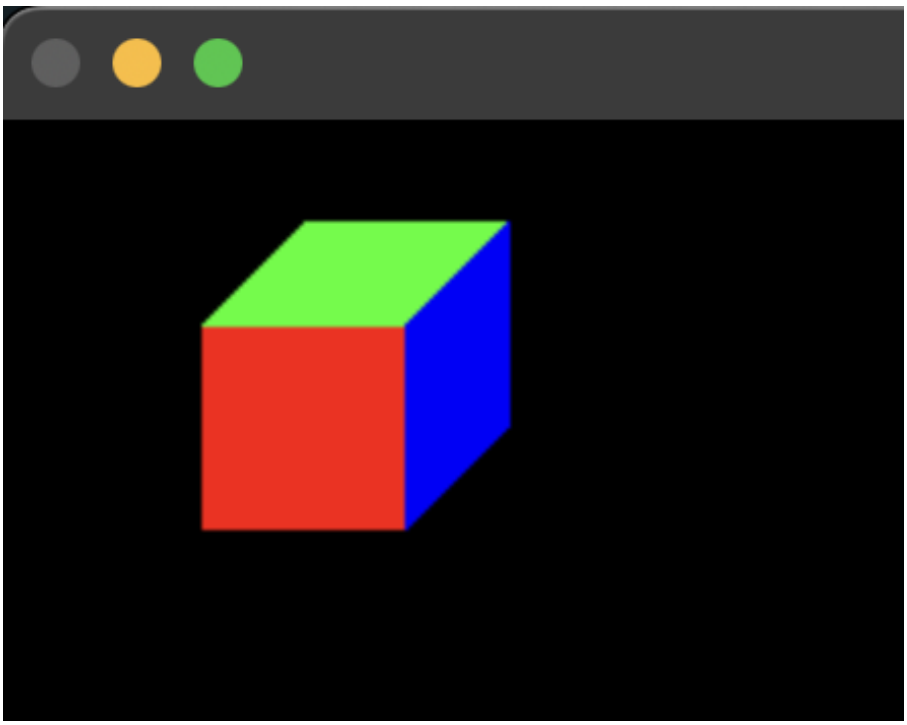


Figure 6.30: Cube rendered manually using OpenCV

The Kinect and OpenCV cube prototype were then combined in order to further prototype a collision avoidance system.

Tuesday, 30 August 2022

Improvements to hand tracking and extracting

Time was spent today on modifying the hand tracking algorithm to work with input from the Kinect sensor and a side-on view as opposed to a top-down approach since this is the desired use case of Mr. Grobler for the overall system - being able to reach into the scene and manipulate virtual objects there. The southmost delta calculations were modified to instead search for the largest region of skin-coloured pixels closest to the centre of the image as this is where the user's hand will predominantly be when interacting with the virtual object. More fine-grained control of the object is going to be necessary and so work turned to the open/closed hand classifier in Tensorflow.

A training data accuracy of 0.9798 and testing data accuracy of 0.9091 could be achieved with 120 input images. Data augmentation was performed and modifications to the network allowed this value to be increased to 1.0 and 1.0 on a small dataset of 240 images. Real-world performance is good and only in small edge cases such as when the hand is rotated to present a small cross seciton of a closed fist does the system classify the geture incorrectly - something that can be alleviated with tweaks to the network size and by introducing more training data. Thus this shows a positive step in using convolutional neural networks for the classifying of gestures and scaling this system up to more gestures is the next milestone.

The OpenCV cube prototype was connected to the improved hand tracking application and the cube could be moved around in the direction reasonably of the user's hand motions. Time was spent also considering how to mount the Kinect on a tripod for use in the final system - although it can actually just sit on top of a tripod the use of zip ties or another material to ensure the Kinect doesn't topple to the ground if the tripod is nudged was also considered.

Wednesday, 31 August 2022

Hand directions neural network

The aim of this session is to build a prototype hand direction classifier.

In order to control the virtual object a model of the hand needs to be identified at each time instance. This need not be a super high-fidelity model as the object just needs to be moved around the screen and rotated in a few directions. Thus building a classifier that can differentiate between a few directions of hand movement might be sufficient to control the object - testing will need to be conducted. A neural network is constructed to predict if a hand is facing left, right or upwards and is visible in Fig. **[6.31]**. The system can accurately classify up and right when using the left hand as input and up and left when using the right hand - the system is learning wrist positions in the background and thus incorrectly classifying gestures when the hand is contorted to point in the same direction as the arm. This will need to be rectified in future implementations.

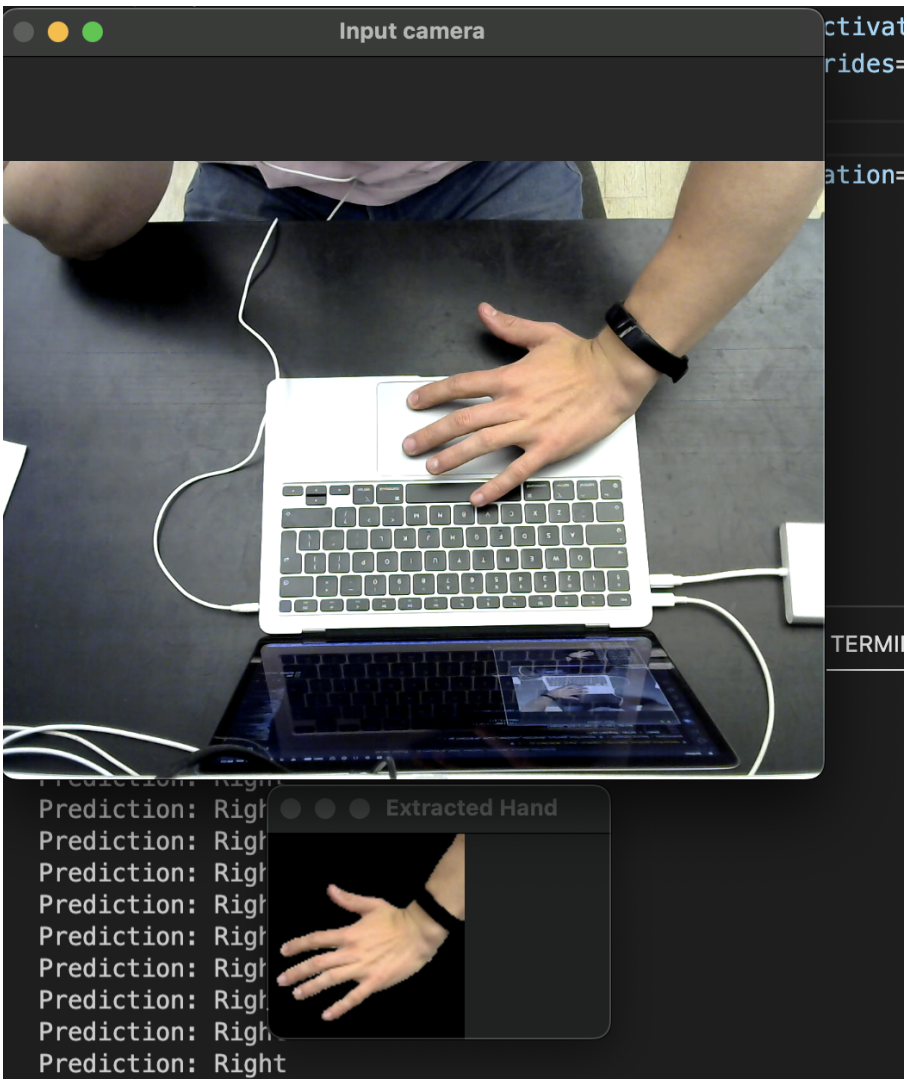


Figure 6.31: Hand direction prediction

In order to have an up to date prototype system the top-down open hand/fist classifier is combined with the OpenGL cube rendering code so that when the user has an open hand nothing happens but when they make a fist and "grab" the object it begins to follow their hand as if they had picked it up. The implementation is visible in Fig. **[6.32]**.

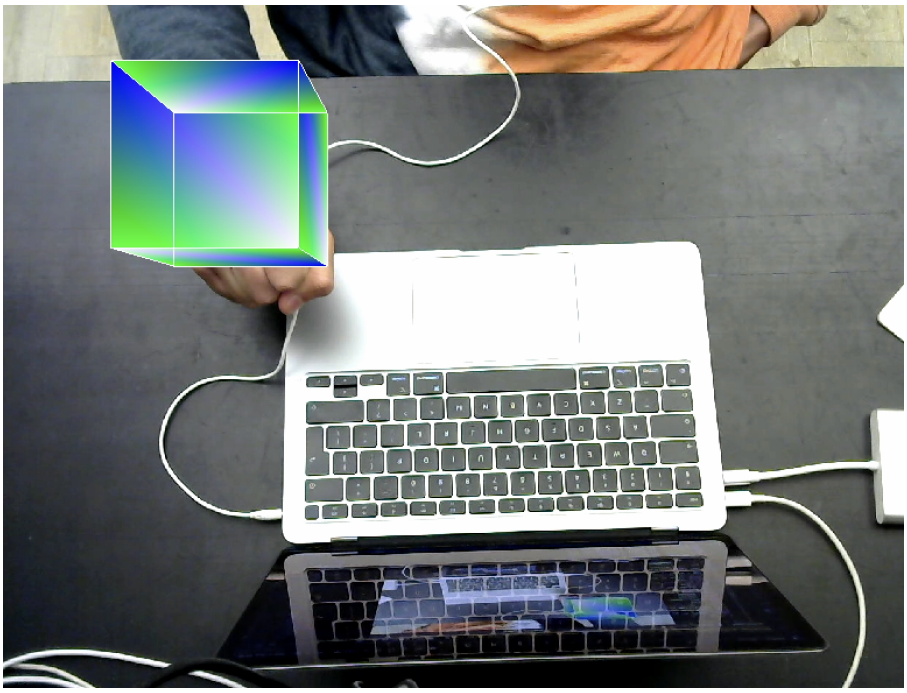


Figure 6.32: Top-down cube grabber prototype

7

September 2022

Wednesday, 07 September 2022

First principles neural network

The aim of this session is to elucidate the recent progress made on the first principles gesture recognition system.

The success of the open hand/fist classifier led to the hypothesis that the virtual object could be adequately controlled by combining the outputs of multiple single gesture/orientation classifiers to build up a model of the current form of the user's hand or gesture. Thus, three classifiers were implemented using Tensorflow - the one specified above that differentiaties between an open hand or closed fist, a classifier that detects whether the palm, back or side of the hand is facing towards the camera and finally if the hand is pointing up, left or right relative to the camera frame - improved upon from an earlier experiment by means of data augmentation. A number of training images were taken, and a neural network modified to learn each of these features. The experiment was a success and saw over 1.0 accuracy for both the open hand/fist classifier and palm/side/back classifier. The direction classifier fared worse with only a 96% accuracy on the test data set - but it is hypothesized that a large amount of input training data could rectify this. Having acquired the model weights using Tensorflow the goal of moving these implementations to a first principles neural network was undertaken.

The neural network built earlier in the year from first principles needed several modifications to achieve the same output as the Tensorflow models utilizing the same architecture. This including changing the activation function from the sigmoid function to the relu function, changing the order and output of both the convolution and maxpooling layer to the same order as the Tensorflow model, changing the Convolution layer's mathematics to allow for multiple colour channels and switching certain operations from laborious for loops to more efficient vectorization solutions - for example, the relu function was changed from an if statement returning 0 if the input value was below 0 and the value otherwise to a matrix multiplication between the entire output matrix and a same-sized matrix of ones and zeroes representing if the value at that particular position is greater than zero or not. This vastly sped up computation. The core of the convolution algorithm is still done with scipy's signal.correlate2d function as the first principles method was just too slow for meaningful inference time. Discussion of this design choice will form part of the final report. In fact, a number of components of the first principles network still need to be vectorized and optimized if real-time gesture recognition is to be performed because the current speed of each inference is 0.3 seconds - only netting a 3 or 4 frames per second output.

These three first principles classifiers were then combined with a slightly modified OpenGL cube program to form a working prototype of the final implementation that allows a user to move their hand close to the cube, close their hand around it and then move the cube across the image frame. Rotating the cube and interacting with the environment around the user is still an ongoing task at the time of writing.

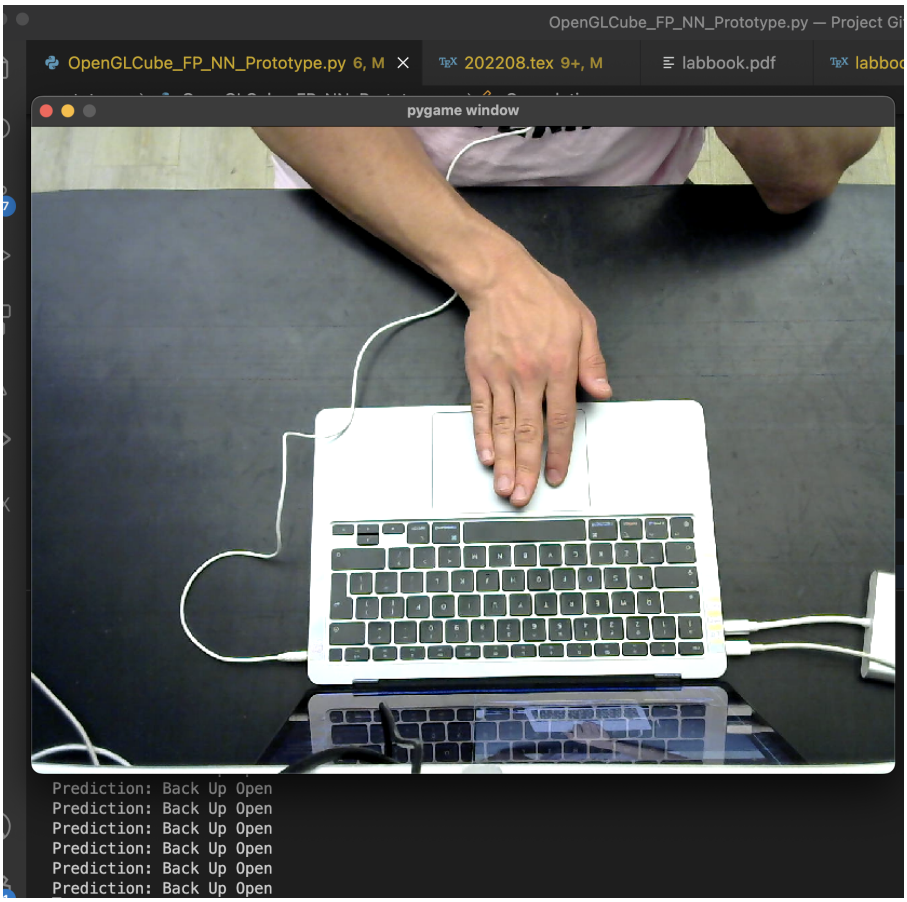


Figure 7.1: Output of first principles gesture classifier

Friday, 09 September 2022

Kinect side-on hand tracking

The aim of this session is to modify the hand tracking algorithm to work from a side view with the Microsoft Kinect sensor.

Although the use of the overhead camera view is useful for gesture recognition, it is challenging to reconcile the coordinate system from a topdown camera and a camera facing the scene from the side. The Microsoft Kinect camera and depth sensor is going to be placed on the side of the user pointing at the desk because this is the view that is most desirable for the creation of the illusion of augmented reality - being able to reach into the scene in front of you and manipulate virtual objects there. However, the hand tracking algorithm was optimized for a topdown view and so needs to be modified in order to track the hand properly through the camera frame.

To this end, an axes detection algorithm was introduced which identifies which edge of the image - top, bottom, left or right, and has the most skin-coloured pixels along it and thus represents the side of the image that the user's hand and arm is entering from. From this information, the largest concentration of skin-coloured pixels closest to the opposite edge of the image can be found and this results in the hand being identified and tracked in the image. Additionally, the size of the extracted image around the hand is modified based on which edge it is closest to so that the full hand and wrist is extracted and sent on the convolutional neural network for further gesture recognition processing. The result is that the hand can be tracked accurately throughout the frame no matter which angle the user enters the frame from and thus the system is even more robust than the topdown hand tracking implementation. The output of the system can be seen in Fig. [7.2].

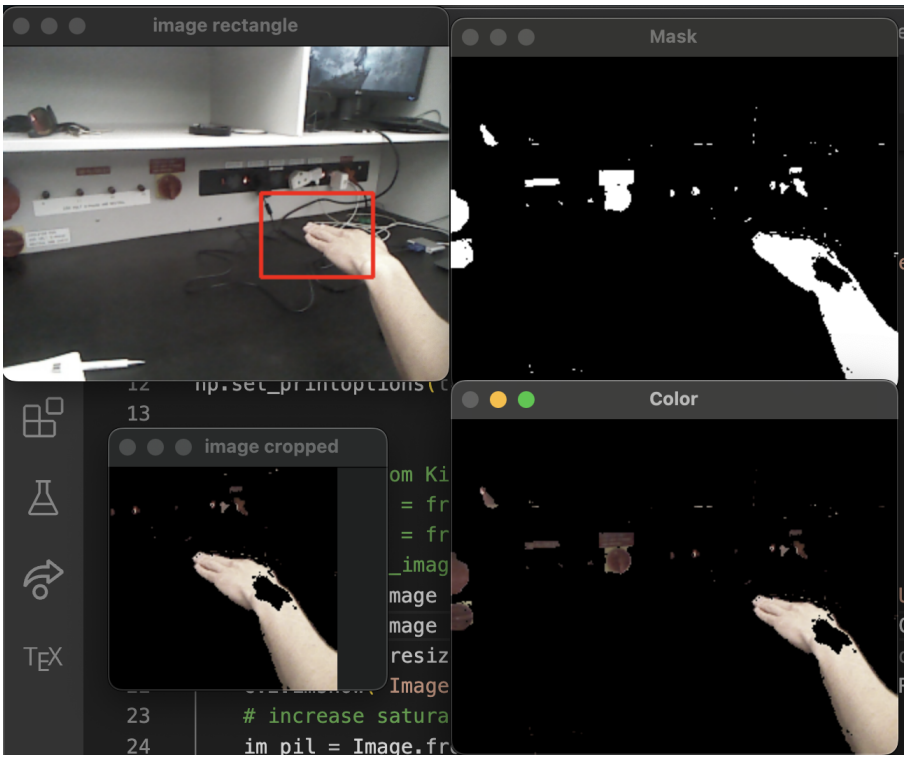


Figure 7.2: Output of the side-on Kinect hand tracking

Monday, 12 September 2022

Speeding up first-principles neural network

In the aid of speeding up the processing of the neural network operations, research was done these past few days on using ctypes or Cython to integrate C code with the Python first principles neural network implementation to speed up operations. Specifically, the use of cytpes to replace the Maxpooling operation was attempted and a Maxpooling implementation in C written. However, problems were encountered when trying to pass a three-dimensional matrix to the C function as the input to the maxpooling algorithm. Work on fixing that integration using pointers and static typing is still ongoing however a more elementary change was introduced to speed up computations. This took the form of reducing the image size the webcam image is resized down to before being passed into the gesture classifier neural networks. Previously the image was resized to 320 x 240 pixels and then processing would take place. This resulted in a large execution runtime for the maxpooling operation. However, after resizing the input image down to just 80 x 60 pixels, the accuracy of the gesture classifiers largely remained the same but the execution time went down from around half a second to only 0.04 seconds as shown in Fig. [7.3]. This is a massive speedup and is highly desirable considering gesture accuracy largely stayed constant. Further testing on just how small the image can be made before significant accuracy is lost will be necessary to further speed up the system.

```
Prediction: Back Right Fist Total runtime: 0.04 seconds
Prediction: Side Right Fist Total runtime: 0.04 seconds
Prediction: Back Up Fist Total runtime: 0.04 seconds
Prediction: Side Right Fist Total runtime: 0.04 seconds
Prediction: Side Right Fist Total runtime: 0.04 seconds
Prediction: Side Right Fist Total runtime: 0.05 seconds
Prediction: Side Right Fist Total runtime: 0.04 seconds
```

Figure 7.3: Output of the OpenGL First principles neural network gesture classifier prototype

Thursday, 22 September 2022

Integrated prototype

Recent changes to the first principles neural network include scaling down the size of the input image to the neural networks in order to increase the speed of inference as well as modifying the hand detection algorithm so that a boolean is returned that represents whether a certain threshold of masked pixels has been detected - whether a hand is present in the frame or not. This can be used later in the algorithmic pipeline in order to keep the cube static and not to run inference on an empty frame.

Three different classifiers are implemented in the first principles integrated prototype and these are the open/closed hand classifier, downwards-facing, side-facing or upwards-facing classifier as well as the forwards-pointing, left-pointing and downwards-pointing classifier. The results of these classifiers are experimentally used to rotate the cube in various manners consistent with how a real-world cube would behave when subjected to the hand motions recognized by the classifiers. This is visible in Fig. [7.4] where the hand is pointing upwards and has rotated the cube in the relevant direction.

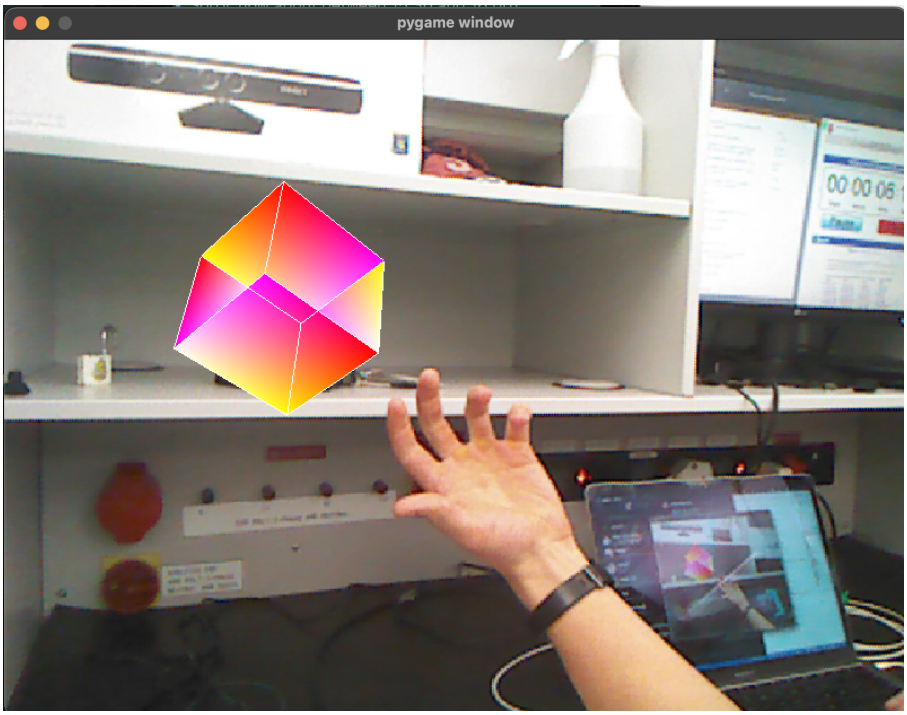


Figure 7.4: Output of the integrated prototype cube rotation

As mentioned previously, the input image is scaled down to 80x60 in the hand segmentation step and this results in much faster execution time for the three gesture classifiers. Specifically, all three classifiers finish their inference in an average of 0.0834 seconds. This is visible in Fig. [7.5] and results in an average frame rate of 12fps. This is half of the required frame rate specified in the project proposal but is predicted to be easily increased by experimentally reducing the size of the neural networks and changing hyperparameters while retaining the current accuracy, which is good and visible in Fig. [7.6] to Fig. [7.8]. The size of the datasets used for these classifiers was 270, 420 and 480 items large respectively. Increasing the size of these datasets and thus the accuracy of the networks and their robustness to noise and incorrect predictions is an ongoing task and will be given particular consideration for the remainder of today's session.

```
Time for inferrance: 0.0834 seconds
Side
Fist
depth_delta:  0.0
Time for inferrance: 0.0825 seconds
Side
Fist
depth_delta:  0.0
Time for inferrance: 0.0822 seconds
```

Figure 7.5: Timers showing the time taken to perform inference on an image for all three classifiers

```
199/199 [==============================] - 3s 13ms/step - loss: 0.5553 - accuracy: 0.7739 - val_loss: 0.4288 - val_accuracy: 0.8261
Epoch 2/5
199/199 [==============================] - 2s 12ms/step - loss: 0.1415 - accuracy: 0.9447 - val_loss: 0.1705 - val_accuracy: 0.9565
Epoch 3/5
199/199 [==============================] - 3s 13ms/step - loss: 0.0512 - accuracy: 0.9799 - val_loss: 0.0229 - val_accuracy: 1.0000
Epoch 4/5
199/199 [==============================] - 3s 13ms/step - loss: 0.0108 - accuracy: 1.0000 - val_loss: 0.0314 - val_accuracy: 1.0000
Epoch 5/5
199/199 [==============================] - 2s 12ms/step - loss: 0.0197 - accuracy: 0.9950 - val_loss: 0.0201 - val_accuracy: 1.0000
2022-09-22 12:01:03.982772: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113] Plugin optimizer for devi
Test loss: 0.3496491611003876
Test accuracy: 0.8958333730697632
```

Figure 7.6: Accuracy of Tensorflow gesture directions classifier on a 270-item large dataset

```
Epoch 2/5
340/340 [==============================] - 4s 12ms/step - loss: 0.1492 - accuracy: 0.9324 - val_loss: 0.0872 - val_accuracy: 0.9737
Epoch 3/5
340/340 [==============================] - 4s 13ms/step - loss: 0.0749 - accuracy: 0.9676 - val_loss: 0.0621 - val_accuracy: 0.9737
Epoch 4/5
340/340 [==============================] - 5s 16ms/step - loss: 0.0261 - accuracy: 0.9912 - val_loss: 0.0111 - val_accuracy: 1.0000
Epoch 5/5
340/340 [==============================] - 4s 13ms/step - loss: 0.0044 - accuracy: 1.0000 - val_loss: 0.0047 - val_accuracy: 1.0000
2022-09-22 12:02:06.676781: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113] Plugin optimizer for devi
Test loss: 0.20805314183235168
Test accuracy: 0.9523809552192688
```

Figure 7.7: Accuracy of Tensorflow gesture down, side, upwards facing classifier on a 420-item large dataset

```
Epoch 2/4
388/388 [==============================] - 5s 13ms/step - loss: 0.1351 - accuracy: 0.9485 - val_loss: 0.1933 - val_accuracy: 0.9318
Epoch 3/4
388/388 [==============================] - 5s 12ms/step - loss: 0.0727 - accuracy: 0.9820 - val_loss: 0.2511 - val_accuracy: 0.9318
Epoch 4/4
388/388 [==============================] - 5s 12ms/step - loss: 0.0352 - accuracy: 0.9974 - val_loss: 0.3153 - val_accuracy: 0.9318
2022-09-22 12:03:17.398591: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113] Plugin optimizer for devic
Test loss: 0.42111214995384216
Test accuracy: 0.8958333730697632
```

Figure 7.8: Accuracy of Tensorflow gesture open/closed fist classifier on a 480-item large dataset

8

October 2022

Monday, 03 October 2022

Final version of literature review

The literature review has been expanded from the original two pages presented in the first semester progress report to five pages. It is presented below for continuity and backup's sake.

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 sub-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 Billinghurst [10] who utilizes a Microsoft Kinect depth and 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 utilize 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 [11] 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 recognize 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 [12] 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 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 [13] 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 recognizing 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 recognized. 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.

These applications receive hand gestures as input and hand gesture control itself can be considered as 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 [14] 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 [15] to recognize 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 [16] relied on classical approaches to hand pose estimation such as using The Continuously Adaptive Mean Shift algorithm to recognize 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-square 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 [13] 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 special hardware or highly specific algorithms that previous implementations required.

A state-of-the-art hand-tracking application created by Google - dubbed Mediapipe Hands [8], 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 [17] 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 normalization 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 [18] 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 [19], and then regress the joints of the hand using a large convolutional neural network based on the RESNET50 architecture [20].

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 localization 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 normalized 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 [21] 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 [22] and Ge [23] also make use of heatmaps of joint coordinates and subsequent fine-tuning algorithms to output joint locations based on the intermediate layers. Ding [22] 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 [23] 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.

Wu [24] 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 [25] inspired convolutional neural network to generate bounding boxes for a detected hand in the input image. The image is then cropped to this bounding box and passed to the skeleton-difference loss neural network. This network seeks to minimize 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 [9] 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 optimized 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 translation. 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.

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 large-scale neural networks 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 localizes 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 and model realistic interactions.

Monday, 10 October 2022

Revised Final version of literature review

Additional information pertaining to plane estimation using depth data was added to the literature review. This was done per Mr. Grobler's instructions at the final individual progress meeting last week.

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 Billinghurst [10] who utilizes a Microsoft Kinect depth and 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 utilize 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 [11] 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 recognize 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 [12] 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 [13] 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 recognizing 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 recognized. 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 [26] utilizes 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. the 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 [27] 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 [27] goes on to estimate planar surfaces from the extracted edges and project virtual shapes onto the surfaces. Other applications such as by Liu [28] utilize 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 [29] with the combination of RANSAC and MDL 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 [14] 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 [15] to recognize 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 [16] relied on classical approaches to hand pose estimation such as using The Continuously Adaptive Mean Shift algorithm to recognize 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 [13] 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 [8], 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 [17] 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 normalization 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 [18] 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 [19], and then regress the joints of the hand using a large convolutional neural network based on the RESNET50 architecture [20].

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 localization 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 normalized 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 [21] 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 [22] and Ge [23] also make use of heatmaps of joint coordinates and subsequent fine-tuning algorithms to output joint locations based on the intermediate layers. Ding [22] 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 [23] 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 [24] 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 [25] 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 minimize 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 [9] 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 optimized 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 localizes one or two hand landmarks will be much cheaper to develop computationally and allow better operation using an embedded device and first-principles algorithms.

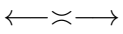
The additional use of pre-processing using skin color and hue segmentation, depth data as well as hand detection and bounding-box algorithms - whether classical or using a deep-learning approach, will also greatly reduce the complexity of the gesture recognition and hand pose estimation design work to 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.

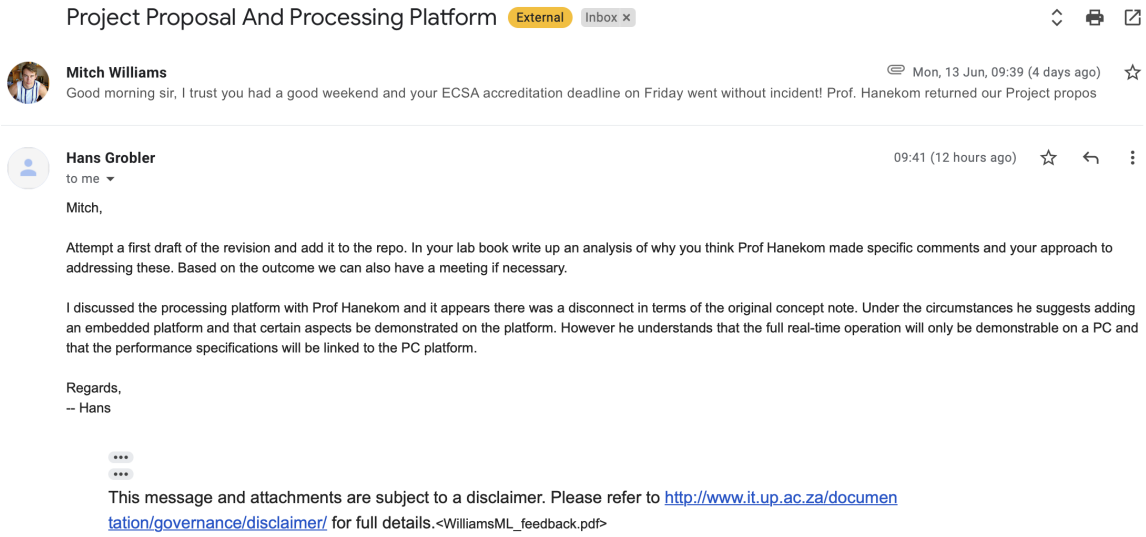
Final implementation progress

Five weeks remain until the final project report is due and so focus will now shift from the system's implementation to completing that report. Work has already begun on it in the form of scaffolding out the section subheadings for the design and implementation section and in the form of the literature review progress above. The surface detection and object detection systems still need to be integrated and the embedded platform implementation actually working to a demonstration-worthy point. Additionally, the neural networks used for hand detection need to be improved and so work will still continue in parallel with the report writing.

Appendix

Acronyms





Project Proposal And Processing Platform External Inbox ×

Mitch Williams Mon, 13 Jun, 09:39 (4 days ago)

Good morning sir, I trust you had a good weekend and your ECSA accreditation deadline on Friday went without incident! Prof. Hanekom returned our Project propos

Hans Grobler 09:41 (12 hours ago)

to me

Mitch,

Attempt a first draft of the revision and add it to the repo. In your lab book write up an analysis of why you think Prof Hanekom made specific comments and your approach to addressing these. Based on the outcome we can also have a meeting if necessary.

I discussed the processing platform with Prof Hanekom and it appears there was a disconnect in terms of the original concept note. Under the circumstances he suggests adding an embedded platform and that certain aspects be demonstrated on the platform. However he understands that the full real-time operation will only be demonstrable on a PC and that the performance specifications will be linked to the PC platform.

Regards,
-- Hans

This message and attachments are subject to a disclaimer. Please refer to http://www.it.up.ac.za/documentation/governance/disclaimer/ for full details.<WilliamsML_feedback.pdf>

Figure 1: Email dated Friday 17 June 2022 between Mr. Grobler and Mr. Williams

Bibliography

[1] Y. Zhang, N. Wadhwa, S. Orts-Escolano, C. Häne, S. Fanello, and R. Garg, "Du2net: Learning depth estimation from dual-cameras and dual-pixels," in *Computer Vision – ECCV 2020* (A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, eds.), (Cham), pp. 582–598, Springer International Publishing, 2020. 5

[2] Y. Shen, S. K. Ong, and A. Y. C. Nee, "Vision-based hand interaction in augmented reality environment.," *International Journal of Human-Computer Interaction*, vol. 27, no. 6, pp. 523 – 544, 2011. 7, 8

[3] 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. 7

[4] M. Kim and J. Y. Lee, "Touch and hand gesture-based interactions for directly manipulating 3d virtual objects in mobile augmented reality," *Multimedia Tools and Applications*, vol. 75, pp. 16529–16550, Dec 2016. 8

[5] T. Lee and T. Hollerer, "Multithreaded hybrid feature tracking for markerless augmented reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 355–368, 2009. 8

[6] 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, pp. 274–280 Vol. 1, 1999. 8

[7] K. Konstantoudakis, K. Christaki, D. Tsiakmakis, D. Sainidis, G. Albanis, A. Dimou, and P. Daras, "Drone control in ar: An intuitive system for single-handed gesture control, drone tracking, and contextualized camera feed visualization in augmented reality," *Drones*, vol. 6, no. 2, 2022. 8

[8] 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. 8, 10, 16, 31, 71, 73, 103, 109

[9] 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*, pp. 1653–1660, 2014. 30, 31, 105, 111

[10] 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. 70, 101, 107

[11] 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*, pp. 539–544, 2011. 70, 102, 107

[12] 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)*, pp. 111–119, Ieee, 2000. 70, 102, 107

[13] 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*, pp. 212–221, 2004. 70, 102, 103, 108, 109

[14] 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. 70, 102, 109

[15] 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. 70, 102, 109

[16] 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. 70, 103, 109

[17] 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. 71, 103, 109

[18] 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. 71, 103, 109

[19] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017. 71, 103, 110

[20] 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*, pp. 770–778, 2016. 71, 103, 110

[21] 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. 71, 104, 110

[22] 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. 71, 104, 110

[23] 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*, pp. 3593–3601, 2016. 71, 104, 110

[24] 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. 104, 110

[25] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014. 104, 110

[26] R. Schnabel, R. Wahl, and R. Klein, "Efficient ransac for point-cloud shape detection," in *Computer graphics forum*, vol. 26, pp. 214–226, Wiley Online Library, 2007. 108

[27] 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*, pp. 1233–1244, 2016. 108

[28] 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*, pp. 4450–4459, 2019. 108

[29] 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, pp. 95–104, 2010. 108