Hand Gesture Recognition

Samuel Oncken, Steven Claypool

CONCEPT OF OPERATIONS

REVISION - 3
3 October 2022

CONCEPT OF OPERATIONS FOR Hand Gesture Recognition

TEAM <72>

APPROVED BY:

Samuel Oncken 10/3/2022

Project Leader Date

Prof. Kalafatis Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	9/15/2022	Samuel Oncken Steven Claypool		Draft Release
2	9/28/2022	Samuel Oncken Steven Claypool		Revision for FSR Release
3	11/28/2022	Samuel Oncken Steven Claypool		Revision for Final Report Release

Table of Contents

List of Figures

1. Executive Summary

To create large training sets for neural networks, our solution is to create a virtual environment in Unity as opposed to using real humans and equipment. The virtual environment takes as input a real hand gesture dataset recorded by a user with the Leap Motion Controller, applies the gestures to hundreds of diverse human models from MakeHuman (imported into Unity), and records images and videos to build a virtual hand gesture training set. This virtual training set requires significantly less time and personnel and additionally will train the neural networks to similar if not improved gesture recognition accuracy when compared to existing real gesture training sets. Using our virtual environment to create synthetic gesture datasets, users can quickly and easily build reliable training sets tailored precisely to their application.

2. Introduction

Building neural network training sets for hand gesture recognition takes significant time and personnel to get diverse and comprehensive data. Thousands of images or videos must be taken manually to build a training set for a new set of hand gestures. To bypass this, we will create a virtual environment that builds training sets from gesture data recorded by one individual that is applied to hundreds of diverse human models and recorded. The neural networks would show similar accuracy in hand gesture recognition with the virtual training sets, potentially replacing the need for “real” training sets. This would notably expedite the creation of new training sets and simplify the entire process.

2.1. Background

Many scholarly articles covering the concept and feasibility of computer vision have been written in the last decade. However, it is no longer an argument that artificial intelligence and machine learning are here to stay. Through our research of the related work done in the field of gesture recognition, we have recorded a wide variety of methods to bring the idea of computer vision into light. In our project, we will not be focusing on the algorithms used in the creation of neural networks such as Convolutional Neural Networks (CNN) [1,2], but instead we will aim to uncover how the construction of a training set can alter the accuracy of a gesture recognition neural network.

Many of the datasets we have found online, summarized by [3], require a handful of real subjects to perform similar actions in front of a motion sensor (Kinect, Wii remote, etc.) which is recording from a set location. By using a number of different human hand models for the same gesture, some variance between gestures is collected which is required given that a gesture recognition system must not only recognize one size or shape of hand for it to function properly. Other datasets like the American Sign Language Lexicon Video Dataset and those derived from it [4] record human subjects using synchronized cameras all recording from different angles. Once again, this is beneficial to the machine learning process because in practice, a system will not always be looking at a human from a single angle.

Our research is aimed at bringing these important considerations together by generating a “virtual” dataset. By virtual, we mean that instead of having a set number of human subjects come into a studio to record movements, we will be recording the movements of one human (under various conditions such as lighting, distance from sensor, etc.) and applying those movements to randomly generated 3-D human models within the Unity game engine environment. After the virtual human model performs the gesture, we will record images and videos from several virtual camera angles to ensure that our training set aids in the recognition of a gesture from all directions. Similar research has been done using a virtual train set of hand gestures[5]. We are looking to expand upon this student’s research because creating a virtual training set is a much more cost-effective method of gathering data that still produces high accuracy results, which we are seeking to prove when testing our virtual datasets with state of the art hand gesture recognition neural networks.

A major obstacle in human gesture recognition is that reliable datasets are limited in the number of gestures they contain when compared to the vast number of gestures humans use every day. Our solution makes it fast and easy to create entirely new datasets consisting of application specific gestures, resulting in much more diverse gesture sets available for use and reducing the initial limitation.

2.2. Overview

We will design a virtual environment that can be used to create training sets for hand gesture recognition neural networks. This virtual environment will be implemented through Unity and will be able to map hand gestures recorded through the Leap Motion Controller onto randomly generated human models. We will be recording the gestures from a first-person point of view, mounted specifically on the head or chest of the individual performing the gestures.

The Unity environment will have a script that places virtual cameras at random locations in front of the human model performing the gesture (3rd person point of view) to collect varying angles of each gesture. The final script written for the Unity environment will generate and import a random human model, apply a randomly chosen "animation" from the real hand gesture database to the model, randomly place multiple virtual cameras in front of the model, then record and store the images taken as members of the final train set for the performed gesture. Finally, once the train set is produced, we will apply it to a benchmark, state of the art neural network for hand gesture recognition and analyze the accuracy achieved and compare it to the results using a standard, real gesture datasets. In comparing the recognition accuracy of a real gesture training set versus our synthetic (virtual) dataset, we will recognize whether using a virtual training set is a viable and effective method to train a hand gesture recognition neural network. We plan to create two separate training sets (comprised of different gestures) to train against two separate neural networks for increased confidence of results.

2.3. Referenced Documents and Standards

[1] J. Nagi *et al*., "Max-pooling convolutional neural networks for vision-based hand gesture recognition," *2011 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, 2011, pp. 342-347, doi: 10.1109/ICSIPA.2011.6144164.

[2] J. Yu, M. Qin, and S. Zhou, "Dynamic gesture recognition based on 2D convolutional neural network and feature fusion," *Sci Rep* 12, 4345, 2022. https://doi.org/10.1038/s41598-022-08133-z

[3] M. Asadi-Aghbolaghi *et al*., "A Survey on Deep Learning Based Approaches for Action and Gesture Recognition in Image Sequences," *2017 12th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2017)*, 2017, pp. 476-483, doi: 10.1109/FG.2017.150.

[4] C. Correia de Amorim, C. Zanchettin, "ASL-Skeleton3D and ASL-Phono: Two Novel Datasets for the American Sign Language," *arXiv:2201.02065 [cs.CV]*, 2022 https://doi.org/10.48550/arXiv.2201.02065

[5] Z. Helton. *VR Hand Tracking for Robotics Applications.* May 2022, Texas A&M Electrical and Computer Engineering 404 Course Archives.

3. Operating Concept

3.1. Scope

The virtual environment can be used to generate various virtual hand gesture training sets for gesture recognition neural networks. These training sets can be composed of images or videos of the hand gestures. The scope of our project is limited exclusively to hand gesture training sets. With some adjustments to the environment, creating training sets for other body parts or full body gestures is possible as well.

3.2. Operational Description and Constraints

To create a virtual training set, the user must make or find a dataset of gesture recordings and import it to the environment. The environment will animate tens to hundreds of diverse human models according to the imported gesture recordings and take images on each to build the training set for a gesture recognition neural network.

The virtual environment to create gesture recognition training sets uses the Unity game engine and MakeHuman software, both of which are open-source. Users can find gesture datasets online but creating a new gesture dataset would require sensor equipment. In our research, we are using the Leap Motion sensor and tracking software which provides the tracking scripts that allow us to directly map bone transformations to virtual human model hands. We will be recording the transformation information including the position, rotation, and scale of each hand/finger component and exporting the files as animation clips, which are then applied to other (imported) MakeHuman virtual models to be used in data collection. Users of our research can use the same equipment or any other tracking technology with the ability to record rigged component transformations.

3.3. System Description

The system to create virtual training sets is made up of multiple subsystems: Gesture Data Collection, Human Model Generation, Model Animation, and Data Capturing/Collection.

The first subsystem is the Gesture Data Collection subsystem. Bone structure data of the hand of one individual is recorded using the Leap Motion Controller to create a dataset of related hand gestures, such as sign language. Once the gesture animations are recorded and stored properly, they are ready to be used in our complete dataset generation virtual environment.

The Human Model Generation subsystem is where virtual human models are created in MakeHuman and exported as .fbx files to be imported into Unity for future use. This system works in tandem with the Model Animation subsystem that uses the gesture recordings/animation clips to animate the models.

Within the completed dataset generation scene of our virtual environment is the Data Capturing/Collection subsystem that takes images of each model that gets generated, compiling the data into a training set. The subsystem uses a script to take the images or videos at randomized angles to collect comprehensive data of each gesture to ensure the neural networks are trained to acceptable accuracies in recognition.

Shown below is a flowchart depicting the relation of each subsystem.

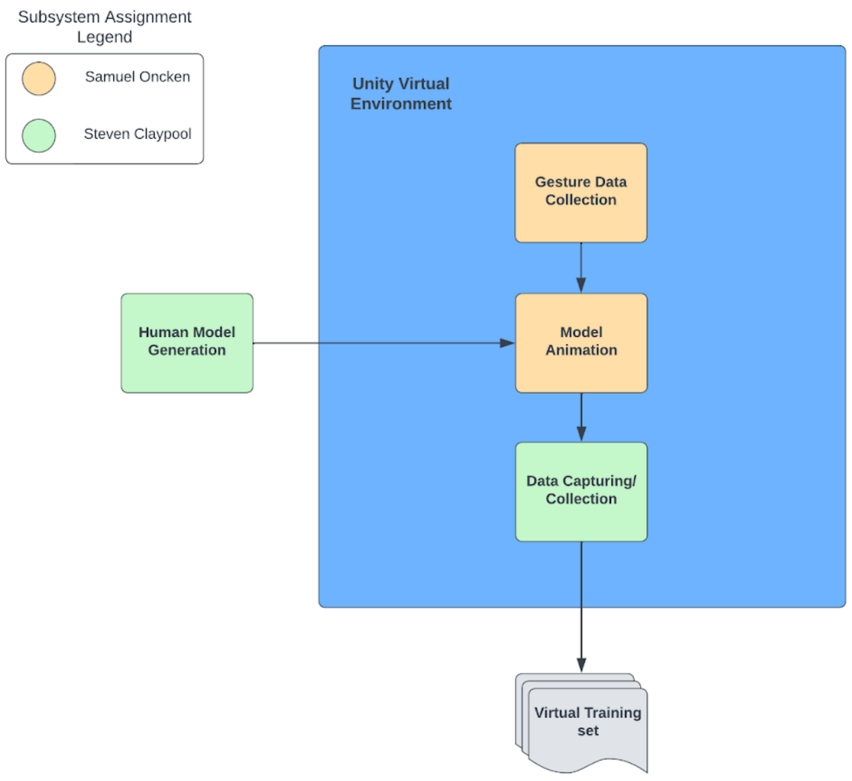


Figure 1: Flowchart of Virtual Hand Gesture Recognition Training Set Generation System

3.4. *Modes of Operations*

Since our project focuses on the validation of virtual datasets for hand gesture recognition neural networks rather than the creation of an operational tool, it does not necessarily have a mode of operation. However, the process of creating a train set using our virtual environment does contain stages: gesture recording and train set generation. During the gesture recording phase, a user must manually record his/her own hand gestures or find an online hand gesture dataset that they want to use. In the train set generation phase, a user will apply their hand gesture data to our virtual environment and the platform will automatically complete the train set using the imported gesture data.

3.5. *Users*

The users of our virtual hand gesture train set generation platform will be anyone who wants to build a dataset for their own applications. We will discuss a few scenarios in the coming section to exemplify this statement. Our environment as well as our tested virtual datasets will be available to download through GitHub. Once we validate through this research that using a virtual environment to create a train set results in as accurate (if not better) recognition than using real humans and equipment, users will be free to apply our platform to any human dataset that they wish to build or expand. They will be required to download Unity software and have some knowledge working within Unity. Users will also be required to be able to create/find a gesture animation dataset to import into our platform as discussed previously, which might require additional software for their chosen sensor (i.e. Leap Motion Controller and Hand Tracking Software).

3.6. Support

Support will be in the form of a written manual which will detail the process of creating a training set using our virtual environment and the Leap Motion Controller. In addition, we will include our gesture animation clips (containing data of the hand gestures recorded with the Leap Motion Controller) and their respective virtual training sets, along with the gesture recognition neural networks we used. A document picturing types of gestures within the datasets will also be included. This can function as a benchmark for the users as they create their own training sets.

4. Scenario(s)

4.1. Software Developers Creating a Game

While creating a game that utilizes sensors such as a VR headset or Kinect, a developer wants to create a gesture dataset of movement controls for a computer vision model to recognize, but he/she does not have the time to find participants or equipment to record the gestures. Instead, the developer will use our virtual train set platform, where he/she will only need one camera to record each gesture. After recording the necessary hand gestures, the developer will apply them to our environment and be able to create an extensive train set with the reliability that a real, time-consuming train set would've produced.

4.2. Sign Language Translation

A user wants to create a dataset to train a model to recognize a distinct/less common form of sign language that does not have an existing dataset. He/she will use our virtual hand gesture train set generation platform to do this. First, the user must record the sign language gestures that he/she wants to implement, which requires only his/her movements (no need for large real human data collection). After importing the collected gesture data into our Unity virtual environment, the user will run our train set generation tool to output a large hand gesture dataset, composed of hundreds of unique human models and photographed from many distinct virtual camera angles. The user can then apply the virtual train set into their sign language recognition neural network.

5. Analysis

5.1. Summary of Proposed Improvements

When compared with the traditional method of collecting "real" gesture data, virtually creating a train set will have a variety of requirements/improvements including:

- Requires only one human to perform each gesture. We do not require paid participants, reducing cost and time for multiple parties.

- Utilizes virtual cameras within the Unity environment, allowing us to store hand gesture data from numerous angles while only recording from one stationary camera location in real life.
- Utilizes the bone/joint location detection within the Leap Motion Hand Tracking software, allowing us to map accurate gesture animations within Unity.
- Applies gesture animations to hundreds of randomly generated human models allowing for the neural network to train using comprehensive data of varying body structures, skin tones, etc.
- Human data recording is not always legal. At a minimum, human data collection requires some amount of paperwork for each participant. Using our research, we reduce legal risk.

5.2. Disadvantages and Limitations

Within our project, we can think of a few potential limitations that include:

- There is one person performing a gesture, then the recorded animation is applied to different human models. As a result, the recognition accuracy for a certain hand gesture might change depending on how a person may perform the gesture differently from others.
    - This can be mitigated by recording multiple versions of the same gesture to account for the randomness in the way a gesture can be performed. Another method is to use one or two more participants to perform each gesture. In both methods, a random version of the gesture recording would be applied to the virtual human model at run time.
- The Leap Motion Controller might not accurately depict the hand gesture being performed by a user into Unity. As a result, a user might have to record his/her gestures multiple times or from different angles to provide the best representation of the real gesture before proceeding to take images and videos for the train set.

5.3. Alternatives

Alternatives to our virtual hand gesture train set generation platform include:

- Traditional process as it stands now: creating a training set using “real” data. Different people perform a gesture while being recorded from either a mounted camera or a variety of cameras. Images and videos of the physical actions are then used to train a neural network. Time consuming and resource intensive.
- Different methods to create virtual datasets. We could use a different game engine software, human model generating software, and motion sensor/tracking software for recording gestures.

5.4. Impact

Our project has a significant influence on society in technological advancement and availability.

By validating the process of creating virtual training sets for neural networks in hand gesture recognition, the possibilities for future virtual training sets expands greatly. Developers could use our research to create diverse gesture recognition training sets much more easily than before. They could also use our research to create new virtual environments for different applications beyond hand gesture recognition.

Through our research, the availability of training set data would drastically increase. As more people begin to use our virtual environment to create training sets, more diverse training sets of certain gesture datasets will be readily available online without needing to find the money and participants to gather data of real individuals. This will also boost the technological advancement of machine learning and AI as more research with training neural networks can be done on much lower budgets.

In addition, our research has certain legal impacts as mentioned previously. Without the need for large amounts of human participants, the creation of a virtual human training set requires far less paperwork and legal permissions to use private data of the participants since only one human is required to perform each gesture that will eventually be applied to a large number of virtual humans.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

FUNCTIONAL SYSTEM REQUIREMENTS

REVISION - 2
3 October 2022

FUNCTIONAL SYSTEM REQUIREMENTS
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72	10/3/2022
Author	Date

APPROVED BY:

Samuel Oncken	10/3/2022
Project Leader	Date
John Lusher, P.E.	Date
T/A	Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	10/3/2022	Samuel Oncken Steven Claypool		Draft Release
2	11/28/2022	Samuel Oncken Steven Claypool		Revision for Final Report Release

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Purpose and Scope

Collecting the large amount of gesture training set data required to teach a hand gesture recognition neural network is time consuming and resource intensive. Our aim is to establish a new method of training set generation using virtual data that takes significantly less time, human participants, and money than the traditional routine. Through our research, we shall develop a platform that takes as input any user recorded gestures (through an LMC) and outputs a full training set, consisting of hundreds of pictures of each gesture (taken from random camera angles) on unique virtual human models.

Additionally, we seek to understand whether a virtual training set can teach a hand gesture recognition neural network to similar if not improved performance when compared to real gesture training set. We shall do this by building at least two virtual training sets that include the exact gestures of two existing, real training sets, then comparing the performance metric of each when used in the same two hand gesture recognition neural network.

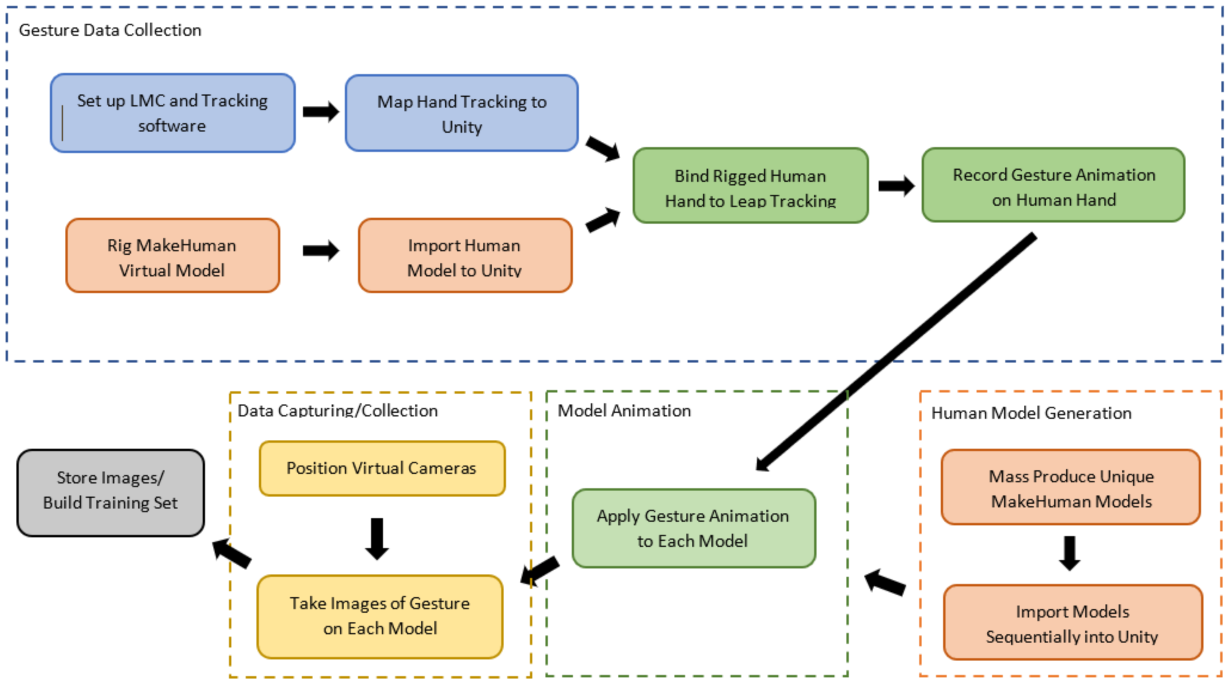

Gesture Data Collection
Set up LMC and Tracking software
Map Hand Tracking to Unity
Bind Rigged Human Hand to Leap Tracking
Record Gesture Animation on Human Hand
Rig MakeHuman Virtual Model
Import Human Model to Unity
Data Capturing/Collection
Position Virtual Cameras
Take Images of Gesture on Each Model
Store Images/ Build Training Set
Model Animation
Apply Gesture Animation to Each Model
Human Model Generation
Mass Produce Unique MakeHuman Models
Import Models Sequentially into Unity

Figure 1. Project Conceptual Image

1.2. Responsibility and Change Authority

The team leader, Samuel Oncken, is responsible for validating that all requirements of this project have been met. If changes to the project are necessary, they will only be carried out after the approval of each team member, project sponsor Professor Stavros Kalafatis, and secondary project sponsor Pranav Dhulipala.

Table 1: Project Subsystem Responsibilities

Subsystem	Responsibility
Gesture Data Collection	Samuel Oncken
Human Model Generation	Steven Claypool
Model Animation	Samuel Oncken
Data Capturing/Collection	Samuel Oncken

2. Applicable and Reference Documents

2.1. Applicable Documents

The following documents, of the exact issue and revision shown, form a part of this specification to the extent specified herein:

Table 2: Applicable Documents

Document Number	Revision/Release Date	Document Title
UH-003206-TC	Issue 6	Leap Motion Controller Data Sheet

2.2. Reference Documents

The following documents are reference documents utilized in the development of this specification. These documents do not form a part of this specification and are not controlled by their reference herein.

Table 3: Reference Documents

Document Number	Revision/Release Date	Document Title
N/A	Version 2021.3 09/23/2022	Unity User Manual 2021.3 (LTS)
N/A	2021	Ultraleap for Developers - Unity API User Manual

2.3. Order of Precedence

In the event of a conflict between the text of this specification and an applicable document cited herein, the text of this specification takes precedence without any exceptions. Any requirements by the sponsor takes precedence over all specifications and documents.

All specifications, standards, exhibits, drawings or other documents that are invoked as “applicable” in this specification are incorporated as cited. All documents that are referred to within an applicable report are considered to be for guidance and information only, except ICDs that have their relevant documents considered to be incorporated as cited.

3. Requirements

3.1. System Definition

The hand gesture recognition system allows users to easily create extensive and diverse hand gesture training sets using a virtual environment. This avoids the need for hundreds of participants for gesture images and videos while still training neural networks to equivalent recognition accuracy. The system has four subsystems: Gesture Data Collection, Human Model Generation, Model Animation, and Data Capturing/Collection.

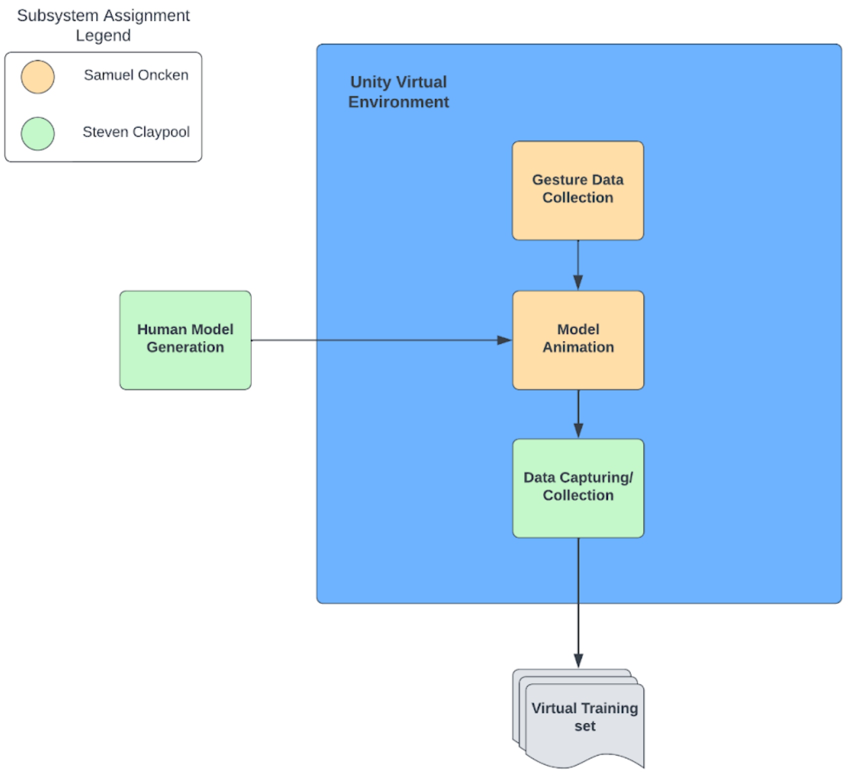


Figure 2. Block Diagram of System

Our Unity environment shall consist of 2 separate scenes. The first scene will house the Gesture Data Collection subsystem, where hand gestures will be recorded using the LMC (mapped to an example MakeHuman model) in Unity and saved as animation clips. Each of these gesture recordings will be accessible to the second scene components and stored to form an animation dataset.

The second scene will house the remaining subsystems. First, the Human Model Generation subsystem will randomly create tens/hundreds of diverse human models and import each sequentially to the Unity environment. This leads to the Model Animation subsystem, which takes a random hand gesture from the animation dataset and applies it to the spawned human model. As this model performs the gesture, the

Data Capturing/Collection subsystem takes images from a randomized virtual camera angles (centered around palm of hand) and compiles the recordings into complete virtual training set folders by name.

3.2. Characteristics

3.2.1. Functional / Performance Requirements

3.2.1.1 Gesture Recognition Accuracy

The metric of performance for the neural network trained with a virtual training set shall be equivalent to the performance metric when using the benchmark training set. We have decided that the gesture recognition accuracy when training using our synthetic data (or combinations of real and synthetic data) shall be no more than 5% below the metric when using the benchmark real training set.

Rationale: For a comparison to be valid, the performance metric of each test must be the same. The gesture recognition accuracy with the virtual training set must be roughly equivalent or better than the accuracy with the benchmark training set to validate the usage of virtual training sets over real training sets.

3.2.1.2. Computer Requirements

The computer running the environment shall use Windows 7+ or Mac OS X 10.7+ with X86 or X64 architecture CPU, a dedicated GPU, 2 GB of RAM, and a USB 2.0 port.

Rationale: Specifications provided by LMC datasheet and Unity 2021.3 manual.

3.2.2. Physical Characteristics

3.2.2.1. LMC Mounting Position

The LMC will operate in a head mounted position to record hand gesture motions.

Rationale: We tested various LMC mounting positions and found that it most accurately records the hand structure data of gestures when mounted on the head directed at the hands.

3.2.2.2. Head Mounting

The head mounting apparatus shall be able to hold 32g of weight.

Rationale: The LMC weighs 32g, which is the only item being mounted.

3.2.3. Electrical Characteristics

3.2.3.1. Input

3.2.3.1.1 LMC Power Input

The LMC requires a 5V DC input with a minimum of 0.5A via USB.

Rationale: Specification provided by LMC datasheet.

3.2.3.2. Output

3.2.3.2.1 Data Output

The system shall output a complete virtual hand gesture training set, composed of images in the file format of .jpg. Images can also be stored as .png files, but that must be changed in the code of the SceneController script. The input file format of the used gesture recognition neural network must be equivalent to the data output produced by our system.

3.2.3.3. Connectors

The Hand Gesture Recognition system shall use a USB-A to Micro-b USB 3.0 cable to connect the LMC to the computer.

3.2.3.4. Wiring

Not applicable to our research.

3.2.4. Software Requirements

3.2.4.1 Neural Networks with TensorFlow 2.8.0

The neural networks used shall work with TensorFlow

Rationale: TensorFlow is a free, open-source software library commonly used in the training of neural networks. We were instructed to use TensorFlow by co-sponsor due to its high accessibility, ease of use, and large support network.

3.2.4.2 Virtual Environment - Unity

The virtual environment used shall be any version from Unity 2021.3 and newer.

Rationale: Unity is a free game engine software that allows for easy use of scripting within the created environments. Additionally, Ultraleap provides Unity specific plug-ins and documentation/support for the use of the LMC hand tracking software.

3.2.4.3 MakeHuman

The MakeHuman software shall be the latest stable version (1.2.0)

Rationale: MakeHuman offers a mass produce function that allows us to randomly generated human models. We will be importing models with a Default *rig, which includes all hand and finger bones for the hand gesture animation to run. We will be using the latest stable version to avoid any potential incompatibility.*

3.2.4.4 Leap Motion Controller

The Leap Motion Controller shall use Gemini - Ultraleap's 5th Generation Hand Tracking software

Rationale: This is the latest version of hand tracking software offered by Ultraleap. It offers the highest quality tracking data and large amounts of usage/support documentation.

3.2.4.5 Unity Plugins

The Ultraleap, Animation Rigging and Unity Recorder Unity plugins are needed to create and run the Unity virtual environment.

Rationale: Ultraleap Unity Plugin includes required scripts for hand tracking/binding as well as useful prefabs of fully rigged hand models for testing. The Unity Recorder is used in the gesture recording process to track bone transform data and export the animation as a usable animation clip file for application on future models. The Animation Rigging package offers a Bone Rendering option which displays bone position on the character model, allowing for easy manipulation of transform data and access to specific bones.

3.2.4.6 MakeHuman Plugin

The MakeHuman "Mass Produce" plugin is needed for large-scale model generation.

Rationale: The plugin is necessary as a means to automatically mass produce diverse human models for animating in the Unity environment.

3.2.4.7 Image Classifiers Python Package

The latest version (1.0.0b1) of PyPI's image-classifiers python package is needed for loading pre trained neural networks for use.

Rationale: The Keras API in TensorFlow is missing some useful image classification models that PyPI's package has, including models we used such as ResNet18.

3.2.5. Environmental Requirements

3.2.5.1 Lighting

There shall be adequate lighting when recording hand gestures with the LMC.

Rationale: The LMC loses gesture recording accuracy in darker conditions, reducing virtual hand gesture quality and therefore decreasing gesture recognition neural network accuracy.

In the virtual environment, lighting shall be manipulated and randomized while performing and recording animations.

Rationale: By varying lighting conditions, we can achieve higher gesture recognition accuracy for more edge cases in testing where images might be darker or brighter than expected.

3.2.5.2. LMC Operating Conditions

The LMC Shall be operated in consideration of the following constraints.

Rationale: Operating conditions are provided by the LMC datasheet.

3.2.5.2.1 Temperature

The LMC shall be operated from 32° to 113° F according to the dataset. This will simply be room temperature in our case as we will be using the LMC in a controlled environment.

3.2.5.2.2 Humidity

The LMC shall be operated at a humidity between 5% to 85%. Again, we will operating the LMC in normal controlled room conditions, which on average is around 35% humidity.

3.2.5.2.3 Altitude

The LMC shall be operated at altitudes between 0 to 10,000 feet. We will be operating the LMC in College Station, which is well within this range.

3.2.6 Failure Propagation

Not applicable to our research

4. Support Requirements

Users of the virtual environment to build virtual hand gesture training sets will require a computer with a gigabyte of storage (more or less depending on the size of the training set being built) and enough computational power (CPU and dedicated GPU) and a display to run the system well. Users must provide power to the computer. The virtual environment is provided along with all the scripts necessary to run the subsystems. A sample neural network and training set will also be provided to act as a benchmark. Any technical issues should be resolved by referencing any specification datasheets, manuals, or by contacting the software companies directly.

Appendix A: Acronyms and Abbreviations

LMC	Leap Motion Controller
CPU	Central Processing Unit
GPU	Graphics Processing Unit

Appendix B: Definition of Terms

Transform data	Data describing the position, rotation, and scale in the x, y, and z directions of a game object within Unity. Rigged models contain parent/child components, which means the child transform is relative to the transform of the parent. For example, the tip of the index finger is a child of the middle bone within the index finger.
Game Object	Building blocks of a Unity environment. These are blank slates which can be characters, objects, or environments that can be manipulated by adding components such as scripts for actual functionality to occur.
Prefab	A fully configured game object that includes specific components/settings that can be stored and reused in scenes or projects. MakeHuman models and Ultraleap-provided rigged hands are examples of prefabs.
Rigged Model	Any model that contains an internal structure that defines its motion. The MakeHuman virtual models are imported with a default rig, defining their skeletal structures with over one hundred unique bones.
Plugin	Software components that add functionality to an existing system. In Unity, we are using multiple plugins that allow us to map rig structures more easily, use prefabs and pre-written Ultra Leap hand tracking scripts, record gesture motion, and export animations in the correct file type.
Neural Network	Combinations of layers and algorithms that operate similar to a traditional biological definition of the human brain in order to recognize patterns and relationships between large amounts of data.
Virtual Environment	Within our research, a virtual environment is defined as the representation of real-world features such as humans and structures from a virtualized Unity project. All humans are computer built models which are animated to resemble real human movement.

Appendix C: Interface Control Documents

Interface Control Documentation is provided in a separate document.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

INTERFACE CONTROL DOCUMENT

REVISION - 2
3 October 2022

INTERFACE CONTROL DOCUMENT FOR Hand Gesture Recognition

PREPARED BY:

Team 72	10/3/2022
Author	Date

APPROVED BY:

Samuel Oncken	10/3/2022
Project Leader	Date
John Lusher II, P.E.	Date
T/A	Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	10/3/2022	Samuel Oncken Steven Claypool		Draft Release
2	11/28/2022	Samuel Oncken Steven Claypool		Revision for Final Report Release

Table of Contents

List of Figures

1. Overview

This document will describe how each subsystem will interface with one another to produce the results discussed in the Concept of Operation and Functional System Requirements. We will walk through the inputs and outputs of each subsystem to fully understand how each build upon the next. Additionally, we will discuss a few physical characteristics of our system.

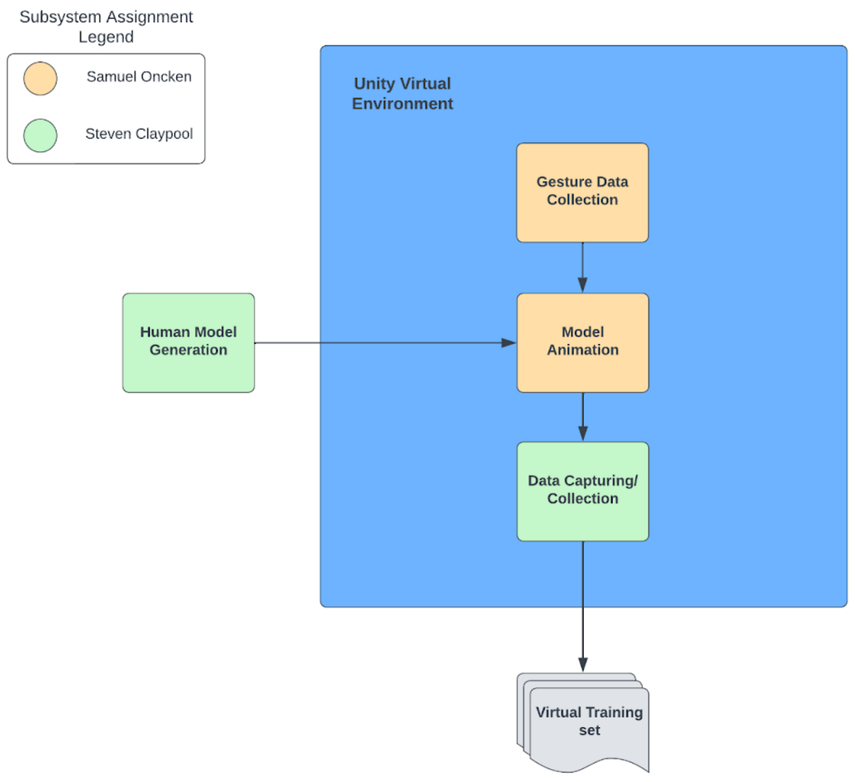


Figure 1: Block Diagram of subsystem interaction

2. References and Definitions

2.1. References

Unity User Manual 2021.3 (LTS)
Version 2021.3
09/23/2022

Ultraleap for Developers - Unity API User Manual
Release date 2021

UH-003206-TC
Leap Motion Controller Data Sheet
Issue 6

2.2. Definitions

LMC	Leap Motion Controller
SOTA	State of the Art
VR	Virtual Reality
FBX	Filmbox File Format
XR	Extended Reality
Spawning	Loads an existing object or model into a scene
Transform data	Data describing the position, rotation, and scale in the x, y, and z directions of a game object within Unity. Rigged models contain parent/child components, which means the child transform is relative to the transform of the parent. For example, the tip of the index finger is a child of the middle bone within the index finger.
Prefab	A fully configured game object that includes specific components/settings that can be stored and reused in scenes or projects. MakeHuman models and Ultraleap-provided rigged hands are examples of prefabs.
Rigged Model	Any model that contains an internal structure that defines its motion. The MakeHuman virtual models are imported with a default rig, defining their skeletal structures with over one hundred unique bones.
Plugin	Software components that add functionality to an existing system. In Unity, we are using multiple plugins that allow us to map rig structures more easily, use prefabs and pre-written Ultra Leap hand tracking scripts, record gesture motion, and export animations in the correct file type.

3. Physical Interface

3.1. Weight

The LMC weighs 32 grams. The cables to connect the LMC to a computer are negligible in weight.

3.2. Dimensions

3.2.1. Dimension of LMC within Gesture Data Collection Subsystem

The LMC is 80mm long, 30mm tall, and 11.30mm deep.

3.3. Mounting Locations

Mounting locations of the LMC for recording hand gesture data include head, chest, screen and desk mounted. From testing, the most accurate recording results from head mounting the LMC, which gives a clear, unobstructed view of the hands and fingers for data collection.

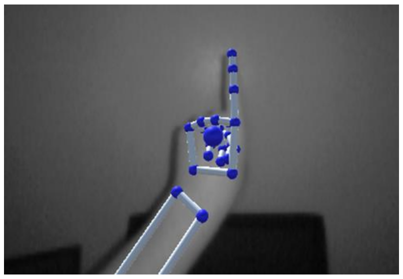


Figure 2: Hand Mapping Through LMC Control Panel

4. Thermal Interface

Not applicable to our research. As mentioned in the FSR, the Leap Motion Controller will function in temperatures from 32° to 113°F, which we will not be exceeding. Therefore, we do not require a thermal interface.

5. Electrical Interface

5.1. Primary Input Power

The LMC needs to receive 5V at a minimum of 0.5A from the USB port.

5.2. Polarity Reversal

Not applicable to our research

5.3. Signal Interfaces

Not applicable to our research

5.4. Video Interfaces

Not applicable to our research

5.5. User Control Interface

Not applicable to our research. User facing interfaces are all within our Unity environment, which is described in the Software Interface section of this report.

6. Communications / Device Interface Protocols

6.1. Wireless Communications (WiFi)

Not applicable to our research. WiFi is required for downloading the necessary software, but once our environment is set up, there is no need for a WiFi connection.

6.2. Host Device

The host device needs a USB 2.0 port minimum for the LMC.

6.3. Video Interface

Not applicable to our research.

6.4. Device Peripheral Interface

The LMC is a peripheral device of our system. Future plans may include VR headsets such as Vive Pro or Oculus headsets.

7. Software Interface

7.1. LMC Interface

7.1.1 LMC Tracking in Unity

The Leap Motion Controller is able to interface with Unity using the “Ultraleap Plugin for Unity” downloaded from the Ultraleap website. This plugin includes a Service Provider (XR) prefab which enables hand tracking and displays transform data relative to the head mounted LMC as shown in Figure 2 below. Additionally, this plugin includes a number of rigged hand prefabs as well as scripts that enable the tracking of each individual bone.

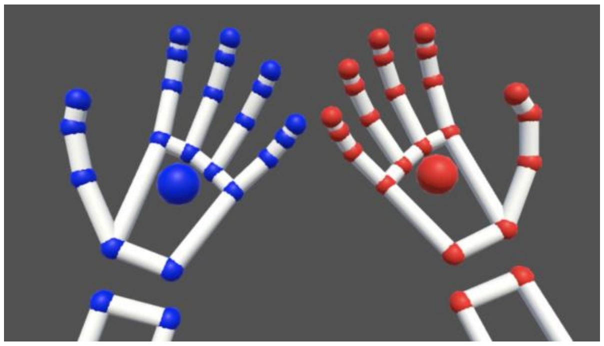


Figure 3: Ultraleap Rigged Hand Model in Unity

7.1.2 LMC Tracking on MakeHuman Model

The LMC must interface with an imported MakeHuman model within Unity for the Gesture Data Collection process. This can be carried out inside of Unity using the Service Provider (XR) prefab as well as the “Hand Binder” script offered in the Ultraleap Unity Plugin. Using this script, we are able to directly map the hand and finger bones of the rigged MakeHuman model to the tracking software, which allows us to display our hand motion on the virtual human.

Rationale: Through our research thus far, we have attempted to record animation clips using the Ultraleap provided rigged hand models then apply the animation to the human model. This worked to some extent, however, the x,y, and z orientations for some bones were different which resulted in incorrect direction of motion on the virtual human. As a result, we found that directly mapping the LMC to the MakeHuman model was more consistent, thus requiring an interface between the two.

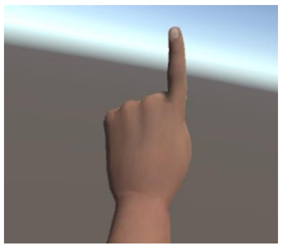


Figure 4: LMC Tracking directly to MakeHuman Model in Unity

7.2. Human Model Interface

7.2.1 MakeHuman Input Interface

For inputting the human models created in the Human Model Generation subsystem into the Unity environment, all models must be set to the default rig and exported as .fbx files. Using the mass produce plugin for MakeHuman, generate and export at least 20 unique virtual models to a “Models” folder within the “Assets” folder of the virtual environment. This will allow the Model Animation Subsystem to access the human model files for spawning and animation within the Unity environment.

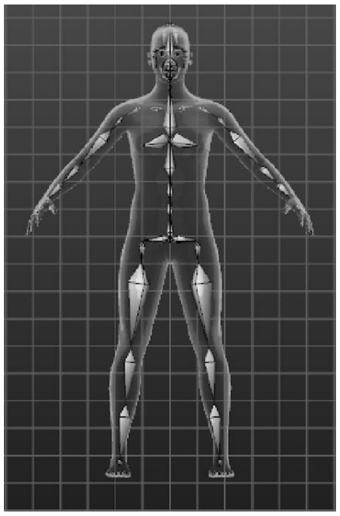


Figure 5: MakeHuman default rig for a human model

7.2.2 Human Model Animation Interface

After gesture recording is completed and a finalized animation dataset is created, it is necessary for the animations to be applied to randomly generated MakeHuman models. This shall be done by adding an “Animator” component to the lower arm bone of the MakeHuman rigged model. The behavior of the animator is determined by an animation controller, where each gesture animation clip is placed. When run, the model will perform the gesture recorded in the animation clip. This will be randomized to apply any given gesture on any given MakeHuman model.

7.3. Unity Camera Interface

The Data Capturing/Collection subsystem works concurrently with the Model Animation subsystem. As human models are animated, multiple cameras in front of the human model take images or videos at various angles. All images are taken as .jpg files.

7.4. Data Storage Interface

The images or videos taken by the cameras in Unity will be organized by dataset and by labeled gesture in the file explorer of the created Unity project. The script will save the full synthetic training set to a designated folder location. Each image will also be labeled according to the naming convention of the real dataset being replicated. For instance, in the Sign Language for Numbers dataset, gesture images of one are labeled as “one” followed by the image number, which is how our system will implement naming as well.

7.5. Full Body Animation Interface

In future integration, our system will be combined with a full body animation environment for a body gesture recognition system. The subsystems of the parent full body gesture system will take precedence over our hand gesture model subsystems except for the Gesture Data Collection subsystem. Our Gesture Data Collection subsystem will record hand gestures that will be spliced onto full body animations randomly. The combined animations will continue through the Model Animation and Data Capture/Collection subsystems of the body gesture recognition system, and the models used will come from the body gesture system's Model Generation subsystem.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

EXECUTION AND VALIDATION PLAN

REVISION - 2
3 October 2022

Change Record

Rev.	Date	Originator	Approvals	Description
1	10/3/2022	Samuel Oncken Steven Claypool		Draft Release for FSR
2	11/25/2022	Samuel Oncken Steven Claypool		Validation Alterations for Final Presentation/Report

VALIDATION PLAN

Table 1: 403 Validation Plan

Test Name	Success Criteria	Methodology	Status	Responsible Engineer(s)
Benchmark Dataset Training	Gesture recognition neural network can run on our home computer and train using real dataset. Results quantified	Download the benchmark dataset and code for the CNN. Run the code and confirm similar accuracy to benchmark logs provided	TESTED – Pass	Steven Claypool
Virtual Dataset Training	Gesture recognition neural network can train using our built dataset and provide accuracy results	Take a final virtual dataset modeled after a real benchmark dataset and use it to train the same CNN as the benchmark. Obtain and compare recognition accuracy to real dataset training run.	TESTED - Pass	Steven Claypool
Gesture Recognition Accuracy	Accuracy is within 5% of benchmark accuracy using real dataset	Train gesture recognition neural network using real and synthetic sets and compare accuracy, testing virtual on virtual, real on real	TESTED - Pass	Steven Claypool
Synthetic Train on Real Test accuracy	Test real data with CNN trained on synthetic data and achieve accuracy similar or improved to benchmark train results	Train a CNN using different methods/compositions with synthetic and/or real data and test using real data	IN PROGRESS	Steven Claypool
Unity Hand Mapping	Real hand movement from LMC is mapped using Ultraleap prefab hand models in Unity	Set up Unity environment, install Ultraleap plug-ins, familiarize with Ultraleap scripts/prefabs, read Ultraleap Unity API documentation, and map hands.	TESTED - Pass	Samuel Oncken
Import a Rigged MakeHuman Model	A fully rigged MakeHuman model is imported into Unity with specific materials/textures extracted	Configure MakeHuman model with “Default” rig and desired appearance. Export model as .fbx file and import model into Unity. Make sure all bones are aligned properly and can be manipulated through space. Approve that model mesh (appearance) is as desired.	TESTED - Pass	Steven Claypool
Virtual Model Unity Hand Mapping	Map hand movement onto an imported rigged MakeHuman model with natural looking movements	Import a rigged MakeHuman model, access the wrist bone and add a hand binder component. Fix advanced issues such as bone orientation and mapping accuracy. Make sure user movement is accurate to movement of the model (without unnatural joint movements).	TESTED – Pass	Samuel Oncken
Mounting Stability	Head mounted LMC remains intact with mounting apparatus	Apply LMC to the mounting apparatus and plug the device into the computer. Rotate head in all directions (looking up, down, left,	TESTED - Pass	Samuel Oncken

	even if head motion is present	right) and shake head left to right. Make sure LMC is still firmly in place.		
Gesture Recording	From within the Unity environment, gesture animation clips can be produced and stored for application on future models	From within the scene used to validate the Unity hand mapping, download the Unity Recorder from the package manager in Unity. Choose output file name and destination, exporting as an animation clip, recorded from the lower arm bone of the virtual model	TESTED – Pass	Samuel Oncken
Apply Example Animation to Model	MakeHuman model is able to perform an imported full body gesture accurately.	Visit Mixamo.com, download a .fbx animation file from the choices listed. Apply the animation to the rigged human model using an Animator component to validate that the rigged structure is able to move as intended.	TESTED - Pass	Samuel Oncken
Apply Recorded Gesture Animation to Model	Freshly imported rigged MakeHuman model is able to perform the gesture animation as recorded in the Gesture Data Collection subsystem	After recording a gesture animation, apply it to a freshly imported MakeHuman model using the Animator component located in the model's lower arm bone	TESTED - Pass	Samuel Oncken
Create and Import Rigged MakeHuman Models to Unity	Minimum 30 MakeHuman models are generated uniquely and imported to Unity environment	Use MakeHuman “mass produce” function to generate unique character models, each fit with a “Default” rig. Produce around 20% of models with edge/corner case metrics (gloves, unnatural colors, etc.)	TESTED - Pass	Steven Claypool
Data Capture Output and File Type	Virtual camera outputs image data as .jpg file type	After images or videos are taken of a model performing a gesture, validate that the images are stored, organized, and are of the desired file type.	TESTED – Pass	Samuel Oncken
Final System Validation	With the press of a button, a large and diverse virtual training set is produced	Run system and validate in output files that each gesture has at least 500 images of gesture performance on differing human models from numerous camera angles	TESTED – Pass	Samuel Oncken

Execution Plan

Table 2: 403 Execution Plan Status Indicator Legend

Complete	🟩
In progress	🟨
Planned	🟦
Behind Schedule	🟥

Table 3: 403 Execution Plan

Task	Deadline	Responsibility	Status
Develop research outcomes/goals	9/7/2022	All	🟩
Get familiar with LMC, Unity, and MakeHuman	9/14/22	All	🟩
Write ConOps	9/15/22	All	🟩
Determine optimal mounting position of LMC with testing of tracking accuracy	9/21/22	Samuel Oncken	🟩
Generate 4 test MakeHuman models, instantiate them into Unity	9/21/22	Steven Claypool	🟩
Map movement of hands onto Ultraleap hand prefabs in Unity	9/28/22	Samuel Oncken	🟩
Add rig settings and animator component to humans in Unity, test online example animation on models	9/28/22	Samuel Oncken	🟩
Set virtual cameras in Unity to face human model upon spawn	9/28/22	Steven Claypool	🟩
Write FSR, ICD, Execution and Validation plans	10/3/2022	All	🟩
Map real hand movements onto virtual human model and record a gesture animation	10/5/2022	Samuel Oncken	🟩
Research Image Synthesis for Gesture Recording process	10/5/2022	Steven Claypool	🟩
Find at least 2 real gesture datasets, tested against SOTA gesture recognition neural networks for us to replicate	10/5/2022	Steven Claypool	🟩
Midterm Presentation	10/12/2022	All	🟩
Record all gesture animations for the replicated virtual gesture datasets	10/12/2022	Samuel Oncken	🟩
Build Unity environment with multiple virtual cameras angled at imported model hand	10/12/2022	Samuel Oncken	🟩

Apply recorded animation to a virtual model	10/19/2022	Samuel Oncekn	🟩
Generate human models using mass produce and export each as a .fbx file to import into Unity	10/19/2022	Steven Claypool	🟩
Find gesture recognition neural network for each dataset and test on machine	10/26/2022	Steven Claypool	🟩
Status Update Presentation	10/27/2022	All	🟩
Complete Unity environment; spawn unique model, apply gesture, record image and store into dataset specific folders	11/4/2022	Samuel Oncken	🟩
Set up Unity environment on WEB 156 lab computer – generate virtual datasets for sign language for numbers and alphabet	11/7/2022	All	🟩
Preprocess training sets and begin training the gesture recognition neural networks using virtual datasets	11/11/2022	Steven Claypool	🟩
Test various compositions of training set data. Combinations of real/synthetic data for NN training, test on more real data	11/18/2022	Steven Claypool	🟥
Train using various compositions of real/synthetic data and test on pictures of our own hands	11/25/2022	Steven Claypool	🟥
Explore future datasets for replication next semester (full body animations/full body images) ex: HaGRID	11/25/2022	All	🟩
Final Demo/Presentation	12/1/2022	All	🟩
Complete Final Paper	12/4/2022	All	🟩

Note: The training/testing process using our virtual data has been much more time consuming and difficult than expected by our sponsor, hence the training being behind schedule. Steven plans to continue training and testing into winter break to explore the most optimal use of our virtual training sets when it comes to gesture recognition accuracy.

Performance on Execution Plan

At the beginning of this semester, we developed an ambitious execution plan with the help of our graduate student sponsor that moved at a much faster pace than we were able to follow. Thus, once again with the help of our sponsor, we had to make alterations to the plan throughout the course of this semester and include work into winter break. We were successful in completing the creation of a Unity environment capable of generating virtual hand gesture training sets for hand gesture recognition neural networks, though it did take longer than expected originally. As a result, training set testing was pushed back and is still in progress today, which is planned to be completed over winter break.

Performance on Validation Plan

Each subsystem within our Unity environment has been validated following our previously shown plan and has passed each test, proving that they are each functioning as we designed. The entire dataset generation system (with all subsystems merged) has also been validated and can replicate real benchmark datasets by capturing images of hand gestures being applied to randomly spawned, unique models under randomized lighting and background conditions and storing each into dataset specific folders sorted by gesture. Validation of the dataset's ability to properly train a hand gesture recognition neural network through testing with real data is still in progress, but we have been able to validate that the virtual data can be used in the training of hand gesture recognition neural networks and achieve >99% accuracy when testing with other virtual data.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

SUBSYSTEM REPORTS

REVISION - 1
28 November 2022

SUBSYSTEM REPORTS
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72	11/28/2022
Author	Date

APPROVED BY:

Samuel Oncken	11/28/2022
Project Leader	Date
John Lusher, P.E.	Date
T/A	Date

Change Record

Rev.	Date	Originator	Approvals	Description
-	11/28/2022	Samuel Oncken Steven Claypool		Release for Final Report

Table of Contents

List of Tables

List of Figures

1. Introduction

The virtual dataset generation Unity environment that we have created to this point has allowed us to fully replicate two real hand gesture datasets of static gestures: American Sign Language and Sign Language for Numbers. The created system is broken down into gesture data collection, human model generation, model animation, and data capturing/collection subsystems. We have tested each subsystem individually across various Unity scenes to validate that all functionality is as we designed and we have merged all subsystems into a common Unity scene to form our complete virtual hand gesture dataset generation system, which has also been validated. We are currently training various gesture recognition models using our virtual datasets either independently or in tandem with the existing real data to evaluate how useful synthetic data can be in increasing gesture recognition accuracy. In addition, we are exploring additional real datasets that include dynamic gestures and full body images/videos to expand upon our research in the coming semester.

Links to each replicated gesture dataset and background image dataset are found below:

American Sign Language: https://www.kaggle.com/datasets/kapillondhe/american-sign-language
Sign Language for Numbers: https://www.kaggle.com/datasets/muhammadkhalid/sign-language-for-numbers
Describable Textures Dataset: https://www.robots.ox.ac.uk/~vgg/data/dtd/

2. Gesture Data Collection Subsystem Report

2.1. Subsystem Introduction

The gesture data collection subsystem is a scene in the Unity project environment where a user can record his/her own gesture animations to be included in the dataset they want to generate. To use this subsystem, the user must own a Leap Motion Controller and have downloaded the necessary tracking software and Unity plug-ins from the Ultraleap website. All instructions for use of this subsystem have been written in the GitHub repository we created to store our project files. The gesture data collection subsystem outputs unique animation clips for future use in the model animation subsystem.

2.2. Subsystem Details

The purpose of the gesture data collection subsystem is to record and store user-created animation clips into the Assets of the Unity project folder. By storing animation clips separately, we can place them into Animation Controllers and therefore play them at random on imported virtual human models in future subsystems. This subsystem is the root of the system as without the gesture data collection subsystem functioning correctly, we would not be able to create uniquely identifiable gestures and therefore would have nothing to animate models with and record images of. Due to this, this subsystem has been tested and changed various times to ensure ease of use and accurate gesture recordings.

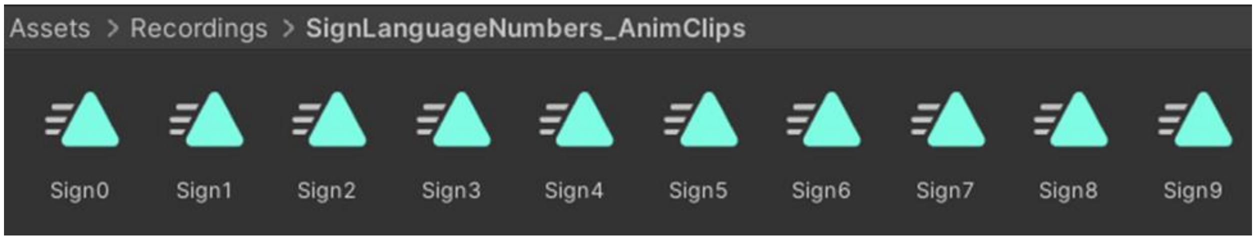
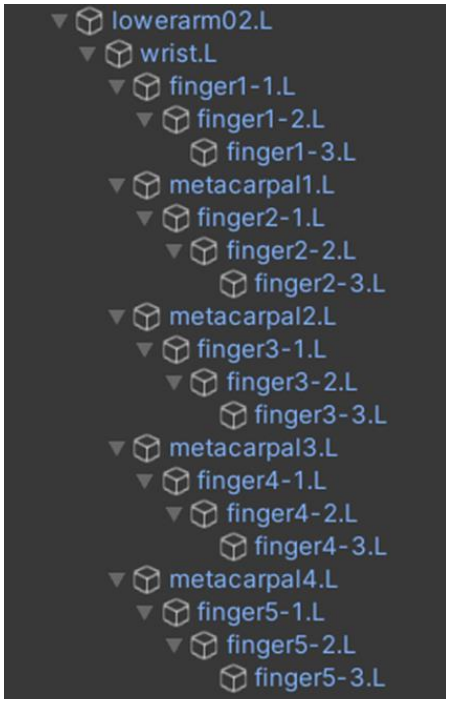


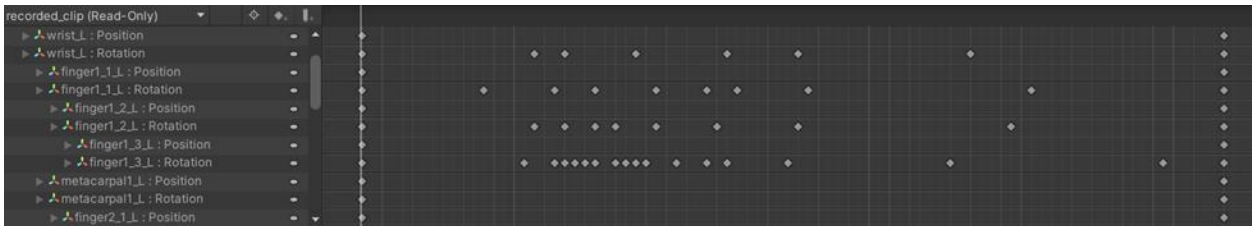
Figure 1: Stored Sign Language for Numbers Gesture Animation Clips

The main challenge with this subsystem has been determining the best method of recording gesture animations while planning for animation application in future subsystems. In the beginning of this project, I had attempted to record gestures (exporting as .fbx files) on the sample hand prefabs provided by Ultraleap. These prefabs contained the necessary scripts already written and configured by the Ultraleap developer team to simplify use. The issue arose when I began recording the transformation data of the hands. Using the Unity recorder, animation clips are simply keyframes of transformation data (position, rotation, and scale) recorded on a fixed interval for each bone of the model. The problem was that the Ultraleap-provided hand prefabs contained bone rigs that were structured and named differently than the MakeHuman models that I was attempting to apply the animation onto. This incompatibility resulted in the animation clips not being able to play on the MakeHuman models. In addition, when exporting the gesture as a .fbx file, the animation clips that came with them contained re-named bones, where each instance of '.' was renamed as '_' (example "wrist.L" turned into "wrist_L"), which once again resulted in naming incompatibility and animation application issues.



```
lowerarm02.L
  wrist.L
    finger1-1.L
      finger1-2.L
        finger1-3.L
    metacarpal1.L
      finger2-1.L
        finger2-2.L
          finger2-3.L
    metacarpal2.L
      finger3-1.L
        finger3-2.L
          finger3-3.L
    metacarpal3.L
      finger4-1.L
        finger4-2.L
          finger4-3.L
    metacarpal4.L
      finger5-1.L
        finger5-2.L
          finger5-3.L
```

Figure 2: MakeHuman Default Rig Bone Labeling



```
recorded_clip (Read-Only)
  wrist_L : Position
  wrist_L : Rotation
    finger1_1_L : Position
    finger1_1_L : Rotation
      finger1_2_L : Position
      finger1_2_L : Rotation
        finger1_3_L : Position
        finger1_3_L : Rotation
    metacarpal1_L : Position
    metacarpal1_L : Rotation
      finger2_1_L : Position
```

Figure 3: Animation clip using FBX Exporter - Not Compatible

The solution to this problem was to map Leap Motion hand tracking data directly to a MakeHuman model (which took more time to configure correctly) and export as an animation clip alone, which resulted in animation clip keyframes that could be directly applied to freshly imported MakeHuman models.

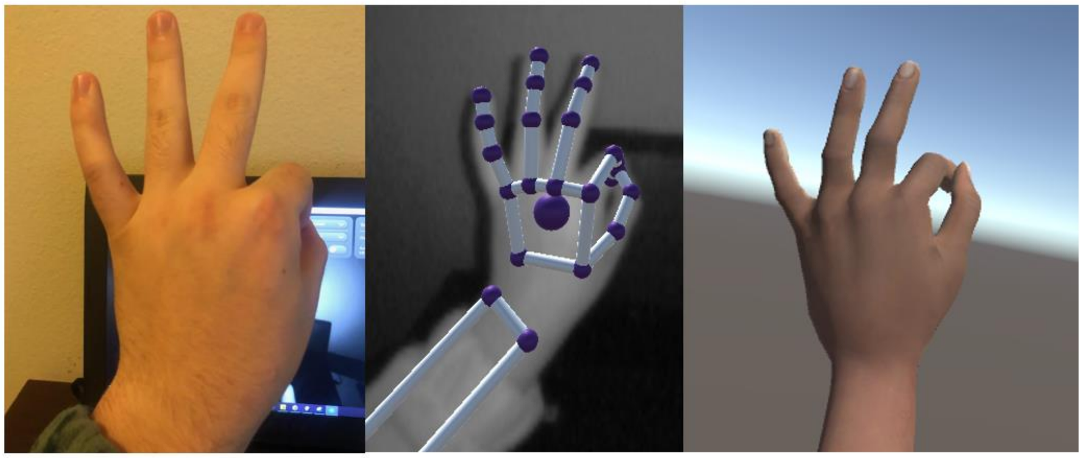


Figure 4: Recording of Animation Clips Directly on MakeHuman Model

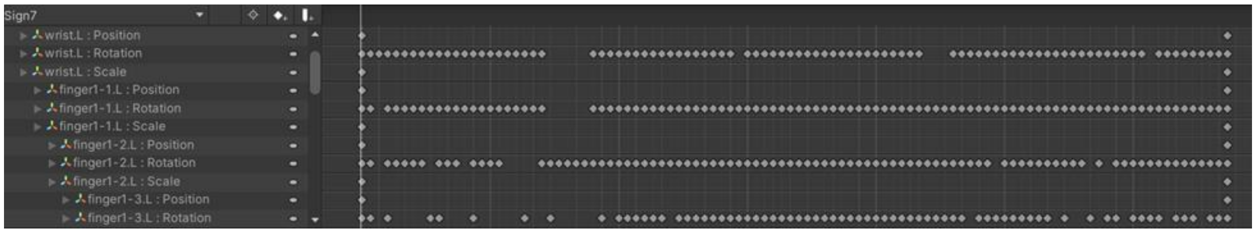
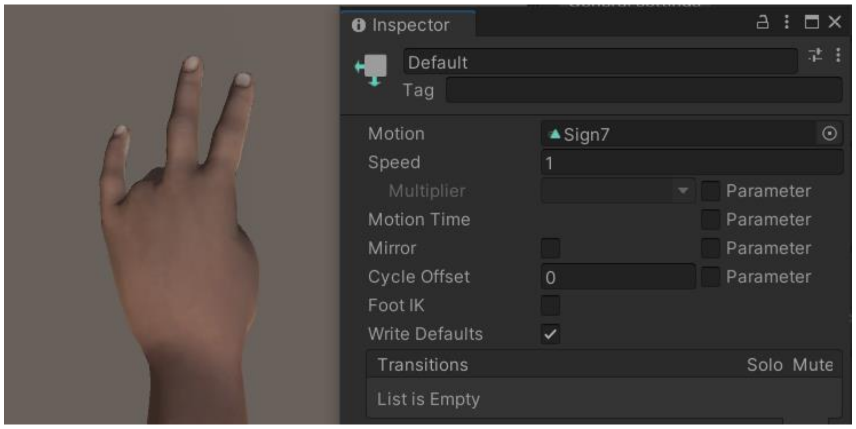


Figure 5: Animation Clip using Finalized Method - Compatible and Accurate

2.3. Subsystem Validation

As the gesture data collection subsystem is the root of other systems, this subsystem was validated by ensuring that the output animation clips were easily accessible and were able to be applied onto imported MakeHuman models in the model animation subsystem. As mentioned in the above paragraph, rigorous testing of different methodologies to record gesture animations were attempted but the final factor in determining which to use was how well the gestures could be applied to a model. To test the model animation application, I created another scene in Unity where I placed a freshly imported MakeHuman model with a “Default” rig and added an Animator component to the “lowerarm02.L” bone of the model, since each animation was recorded from the lowerarm02. I then placed a sample animation clip into an Animation Controller which controlled that Animator component and ran the scene. Once the animation was able to run smoothly and stop in the final hand gesture position as I had recorded, the subsystem proved to be applicable to future subsystems.



Default
Tag

Motion: Sign7
Speed: 1
Multiplier — Parameter
Motion Time — Parameter
Mirror — Parameter
Cycle Offset: 0 — Parameter
Foot IK
Write Defaults ✓
Transitions — Solo Mute
List is Empty

Figure 6: Test Application of Sign7 Gesture Animation Clip

In addition, I validated the use of the Leap Motion Controller within this subsystem by testing various LMC mounting positions under different lighting conditions to see which resulted in the highest tracking accuracy upon inspection. I confirmed that by head mounting the LMC, I was able to achieve the best motion tracking possible for the gestures in datasets that we planned to replicate. Many of the gestures within the American Sign Language and Sign Language for Numbers datasets involved putting one or more fingers down (into palm region) while still holding up others, which I had to ensure the LMC could recreate accurately. In the images below, you can see how finger tracking is easily lost and inaccurate when using the LMC in desktop mounted mode vs. in head mounted mode.

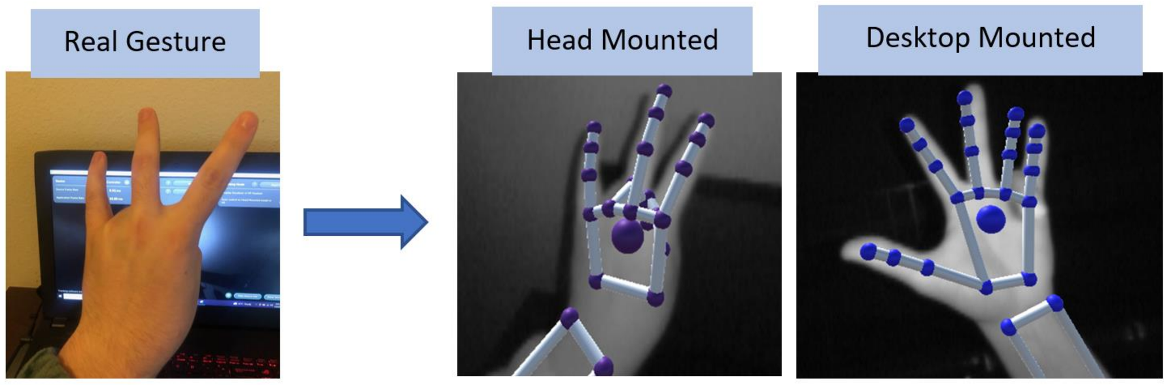


Figure 7: Tracking Accuracy of Head Mounted LMC vs. Desktop Mounted LMC

2.4. Subsystem Conclusion

The gesture data collection subsystem has been tested and validated and has proved to be functioning as intended. This subsystem successfully models tracking data from the user directly onto a MakeHuman model and records and stores the transformation data of the hand bones to usable animation clips that can be applied to future imported models in the subsystems to come.

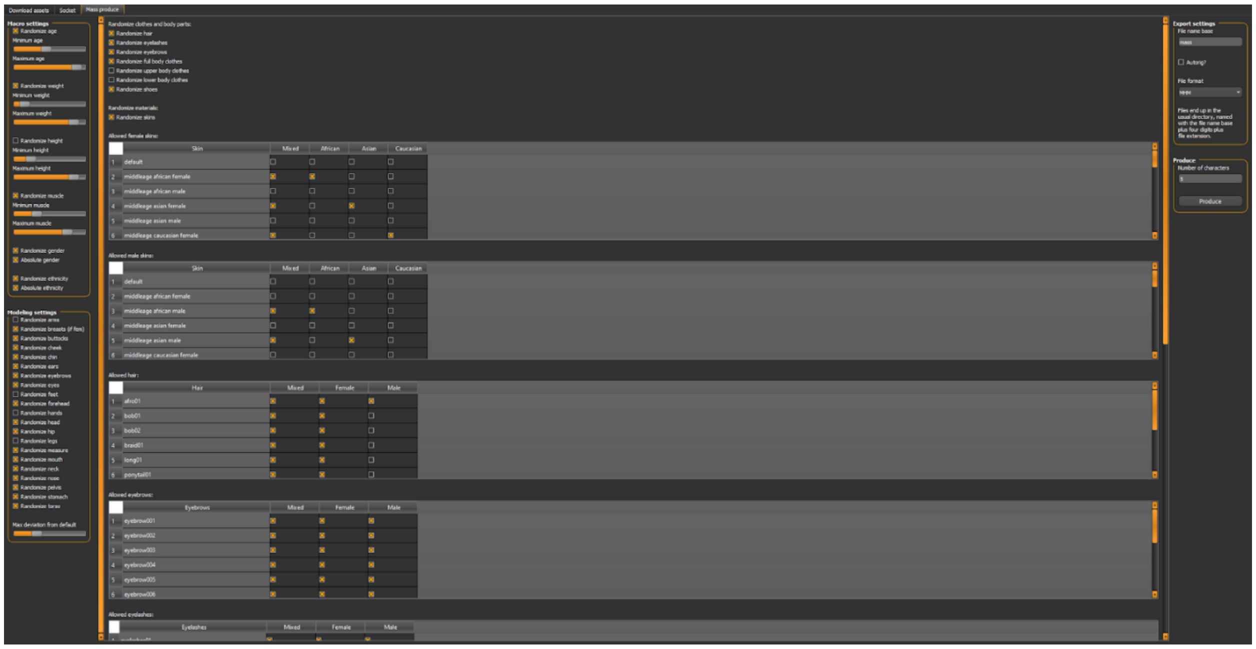
3. Human Model Generation Subsystem Report

3.1. Subsystem Introduction

The human model generation subsystem is responsible for the creation of the human models through the MakeHuman software for gesture animation, as well as the loading and random spawning of the models within the final system's environment. This subsystem works in conjunction with the model animation subsystem, where the created human models (with diverse skin tones, handwear, etc.) are animated in the virtual environment. The subsystem takes input from the user on the number of models to use and loads in the models from the assets folder with names specified in the script function. When the final system is run, the script spawns the model into the scene for the model animation subsystem to manipulate.

3.2. Subsystem Details

Adjustments to the MakeHuman code were originally planned to automate the generation of human models. However, after looking through the Mass Produce plugin and MakeHuman's code and running into multiple issues with the process, we spoke to our sponsor and determined that it would require extensive work to make such changes and resorted to a manual approach since this was outside the scope of our work. The Mass Produce plugin is used to create numerous MakeHuman model files (.MHM extension) that are then reloaded back into MakeHuman. For each model, a default rig is added in 'Skeleton' of the 'Pose/Animate' tab, and handwear/nails are added to 20-30% of the models in 'Clothes' of the 'Geometries' tab. The handwear/nails are not in MakeHuman by default, so the user must go to the 'Download assets' in the 'Community' tab to search for and download the desired textures. The model's heights are all adjusted to roughly 165 cm to ensure the hands of the models are within the frame of the cameras when loaded into the virtual scene. Finally, the model is exported as a FilmBox file (.FBX extension) directly into the model folder within the Unity directory's 'Assets' folder, where the scene controller script can access them.



Macro settings
- [✓] Randomize age
- Minimum age
- Maximum age
- [✓] Randomize weight
- Minimum weight
- Maximum weight
- [ ] Randomize height
- Minimum height
- Maximum height
- [✓] Randomize muscle
- Minimum muscle
- Maximum muscle
- [✓] Randomize gender
- [✓] Absolute gender
- [✓] Randomize ethnicity
- [✓] Absolute ethnicity

Modeling settings
- [ ] Randomize arms
- [✓] Randomize breasts (if fem)
- [✓] Randomize buttocks
- [✓] Randomize cheek
- [✓] Randomize chin
- [✓] Randomize ears
- [✓] Randomize eyebrows
- [✓] Randomize eyes
- [ ] Randomize feet
- [✓] Randomize forehead
- [ ] Randomize hands
- [✓] Randomize head
- [✓] Randomize hip
- [ ] Randomize legs
- [✓] Randomize measure
- [✓] Randomize mouth
- [✓] Randomize neck
- [✓] Randomize nose
- [✓] Randomize pelvis
- [✓] Randomize stomach
- [✓] Randomize torso
- Max deviation from default

Randomize clothes and body parts:
- [✓] Randomize hair
- [✓] Randomize eyelashes
- [✓] Randomize eyebrows
- [✓] Randomize full body clothes
- [ ] Randomize upper body clothes
- [ ] Randomize lower body clothes
- [✓] Randomize shoes

Randomize materials:
- [✓] Randomize skins

Allowed female skins:

	Skin	Mixed	African	Asian	Caucasian
1	default	☐	☐	☐	☐
2	middleage african female	✓	✓	☐	☐
3	middleage african male	☐	☐	☐	☐
4	middleage asian female	✓	☐	✓	☐
5	middleage asian male	☐	☐	☐	☐
6	middleage caucasian female	✓	☐	☐	✓

Allowed male skins:

	Skin	Mixed	African	Asian	Caucasian
1	default	☐	☐	☐	☐
2	middleage african female	☐	☐	☐	☐
3	middleage african male	✓	✓	☐	☐
4	middleage asian female	☐	☐	☐	☐
5	middleage asian male	✓	☐	✓	☐
6	middleage caucasian female	☐	☐	☐	☐

Allowed hair:

	Hair	Mixed	Female	Male
1	afro01	✓	✓	✓
2	bob01	✓	✓	☐
3	bob02	✓	✓	☐
4	braid01	✓	✓	☐
5	long01	✓	✓	☐
6	ponytail01	✓	✓	☐

Allowed eyebrows:

	Eyebrows	Mixed	Female	Male
1	eyebrow001	✓	✓	✓
2	eyebrow002	✓	✓	✓
3	eyebrow003	✓	✓	✓
4	eyebrow004	✓	✓	✓
5	eyebrow005	✓	✓	✓
6	eyebrow006	✓	✓	✓

Allowed eyelashes:

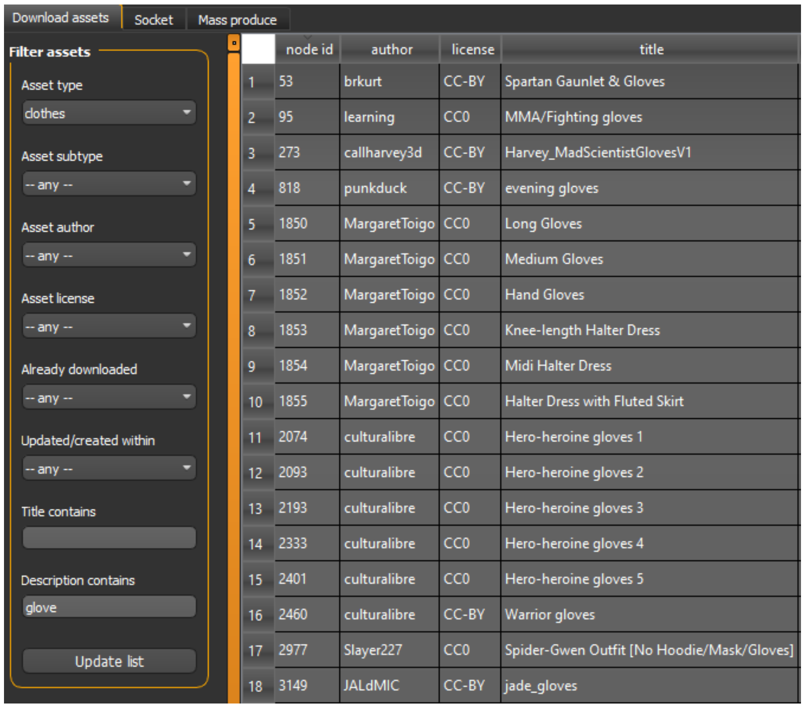
	Eyelashes	Mixed	Female	Male

Export settings
- File name base: mass
- [ ] Autorig?
- File format: MHM

Files end up in the usual directory, named with the file name base plus four digits plus file extension.

Produce
- Number of characters: 5

Figure 8: Mass Produce Plugin Page



Asset type: clothes
Asset subtype: -- any --
Asset author: -- any --
Asset license: -- any --
Already downloaded: -- any --
Updated/created within: -- any --
Title contains:
Description contains: glove

	node id	author	license	title
1	53	brkurt	CC-BY	Spartan Gaunlet & Gloves
2	95	learning	CC0	MMA/Fighting gloves
3	273	callharvey3d	CC-BY	Harvey_MadScientistGlovesV1
4	818	punkduck	CC-BY	evening gloves
5	1850	MargaretToigo	CC0	Long Gloves
6	1851	MargaretToigo	CC0	Medium Gloves
7	1852	MargaretToigo	CC0	Hand Gloves
8	1853	MargaretToigo	CC0	Knee-length Halter Dress
9	1854	MargaretToigo	CC0	Midi Halter Dress
10	1855	MargaretToigo	CC0	Halter Dress with Fluted Skirt
11	2074	culturalibre	CC0	Hero-heroine gloves 1
12	2093	culturalibre	CC0	Hero-heroine gloves 2
13	2193	culturalibre	CC0	Hero-heroine gloves 3
14	2333	culturalibre	CC0	Hero-heroine gloves 4
15	2401	culturalibre	CC0	Hero-heroine gloves 5
16	2460	culturalibre	CC-BY	Warrior gloves
17	2977	Slayer227	CC0	Spider-Gwen Outfit [No Hoodie/Mask/Gloves]
18	3149	JALdMIC	CC-BY	jade_gloves

Figure 9: MakeHuman Page to Download Assets

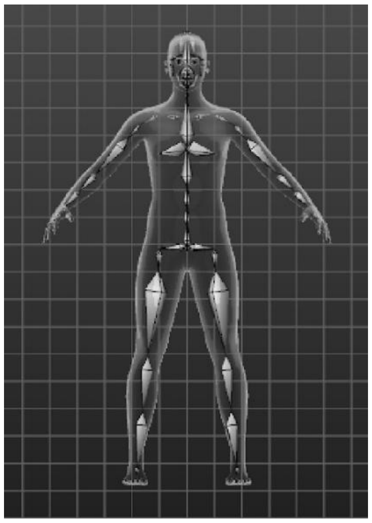
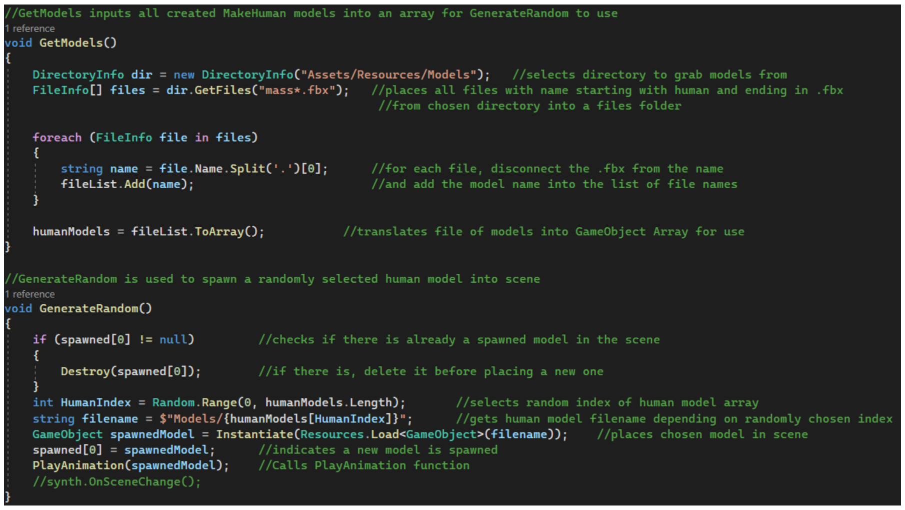


Figure 10: Default Rig for MakeHuman Models

For the loading and random generation of the human models, functions within the scene controller script were written. The model loading script was provided by our sponsor: this script function pulls the models from the Unity Assets and loads each of their names into an array of a size specified by the user. Once in the human model array as elements, the model generation function will destroy any model within the scene first, and will then spawn a random model by index of the array into the scene. The script then runs the model animation subsystem on the spawned model.



```
//GetModels inputs all created MakeHuman models into an array for GenerateRandom to use
1 reference
void GetModels()
{
    DirectoryInfo dir = new DirectoryInfo("Assets/Resources/Models");   //selects directory to grab models from
    FileInfo[] files = dir.GetFiles("mass*.fbx");   //places all files with name starting with human and ending in .fbx
                                                     //from chosen directory into a files folder

    foreach (FileInfo file in files)
    {
        string name = file.Name.Split('.')[0];      //for each file, disconnect the .fbx from the name
        fileList.Add(name);                         //and add the model name into the list of file names
    }

    humanModels = fileList.ToArray();           //translates file of models into GameObject Array for use
}

//GenerateRandom is used to spawn a randomly selected human model into scene
1 reference
void GenerateRandom()
{
    if (spawned[0] != null)         //checks if there is already a spawned model in the scene
    {
        Destroy(spawned[0]);        //if there is, delete it before placing a new one
    }
    int HumanIndex = Random.Range(0, humanModels.Length);       //selects random index of human model array
    string filename = $"Models/{humanModels[HumanIndex]}";      //gets human model filename depending on randomly chosen index
    GameObject spawnedModel = Instantiate(Resources.Load<GameObject>(filename));    //places chosen model in scene
    spawned[0] = spawnedModel;      //indicates a new model is spawned
    PlayAnimation(spawnedModel);    //Calls PlayAnimation function
    //synth.OnSceneChange();
}
```

Figure 11: C# Script Functions for Loading and Spawning Human Models

3.3. Subsystem Validation

The parts of the human model generation subsystem were validated separately as each part was created. The exporting of the models into the Unity Assets folder was first validated, ensuring that the environment had access to the model files that got imported from MakeHuman and that the models could be manually spawned into the scene. Scripts were validated with the newly imported models, testing the loading of the models names into the array as elements and the spawning of the models. The spawned models were observed to ensure that all loaded models could be spawned into the environment sequentially and that they were destroyed at each call of the function. The subsystem itself was also validated in tandem with the model animation subsystem, validating the spawning, animation, and destruction of each model with each call of the script function.

3.4. Subsystem Conclusion

The validation of the human model subsystem is completed and is operating as designed. It takes human models imported from MakeHuman, loads each model name into an array within the Unity environment, and spawns a random model picked from the array into the scene sequentially for gesture animation, after which it reruns the script, destroying the old model and restarting the process with a new random model. This proves the subsystem's function and its incorporation with the related model animation subsystem.

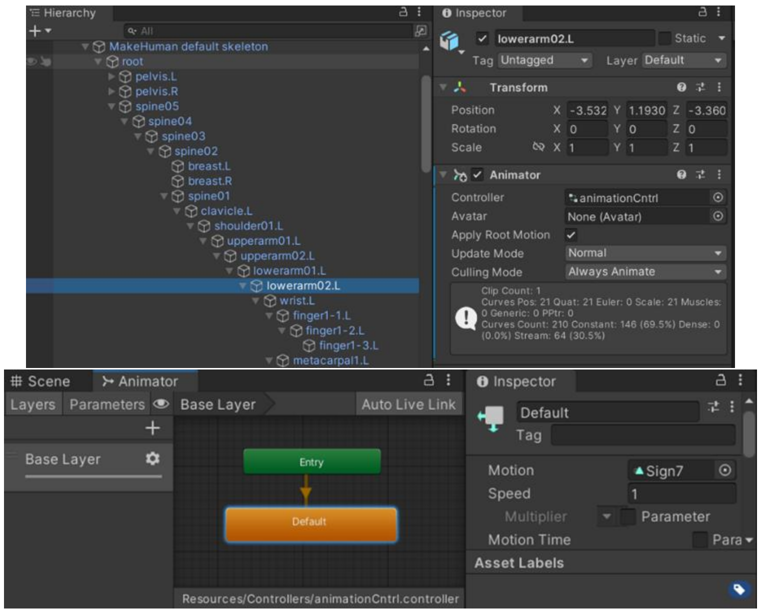
4. Model Animation Subsystem Report

4.1. Subsystem Introduction

The model animation subsystem is responsible for placing the proper animation components on the imported model and randomly selecting an animation to play. This subsystem has been implemented through script and acts only after the gesture data collection subsystem has been completed. This subsystem works in tandem with the human model generation and data capturing/collection subsystems, as once an animation is applied to a randomly spawned human model, an image is taken almost exactly at the same time. The model animation subsystem recognizes which dataset the user is looking to replicate (Sign Language for Numbers, American Sign Language, or a custom-built dataset) and pulls only the animation clips included in that dataset, selecting clips to play until all clips have been displayed a user selected maximum number of times.

4.2. Subsystem Details

As mentioned in the validation process of the gesture data collection subsystem, to play an animation, the model must have an Animator component located on its lowerarm02.L bone as well as a selected Animation Controller, which houses the animation clips that are to be applied to the model. Initially in testing using one additional MakeHuman model, I would simply add an Animator component to the lowerarm02 bone directly, select an Animation Controller for the Animator to use, and add a singular animation clip into the controller to play as shown in the figure below.



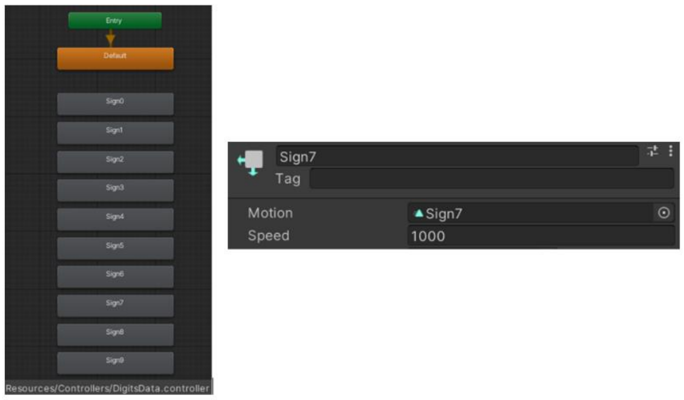
```
MakeHuman default skeleton
  root
    pelvis.L
    pelvis.R
    spine05
      spine04
        spine03
          spine02
            breast.L
            breast.R
            spine01
              clavicle.L
                shoulder01.L
                  upperarm01.L
                    upperarm02.L
                      lowerarm01.L
                        lowerarm02.L
                          wrist.L
                            finger1-1.L
                              finger1-2.L
                                finger1-3.L
                            metacarpal1.L
```

```
lowerarm02.L          Static
Tag Untagged          Layer Default
Transform
Position   X -3.532   Y 1.1930   Z -3.360
Rotation   X 0        Y 0        Z 0
Scale      X 1        Y 1        Z 1
Animator
Controller          animationCntrl
Avatar              None (Avatar)
Apply Root Motion   ✓
Update Mode         Normal
Culling Mode        Always Animate
Clip Count: 1
Curves Pos: 21 Quat: 21 Euler: 0 Scale: 21 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 210 Constant: 146 (69.5%) Dense: 0 (0.0%) Stream: 64 (30.5%)
```

```
Default
Tag
Motion        Sign7
Speed         1
Multiplier    Parameter
Motion Time   Para
Resources/Controllers/animationCntrl.controller
```

Figure 12: Manual Placement of Animator and Controller with One Animation Clip

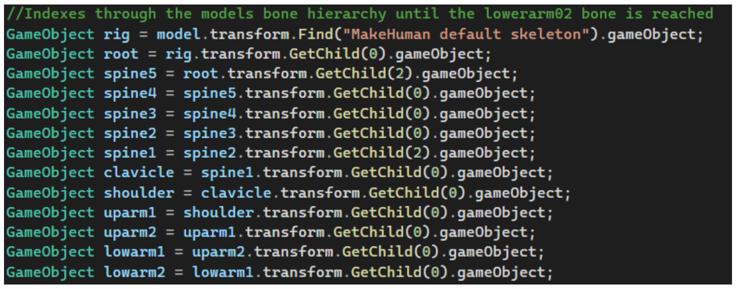
After the Human Model Generation subsystem is implemented, freshly spawned models are being instantiated into the scene (without these components), thus the process of component placement and animation clip selection had to be scripted. But before the Unity scene is run, there must be unique Animation Controllers for each of the datasets being replicated. Each of these animation controllers are labeled according to the dataset they represent and house all animation clips required to build the intended dataset. It was also necessary to speed up the performance of the gesture animations by at least 100x to decrease the time it takes to play all animations required to completely generate a virtual training set.



Sign7
Tag
Motion Sign7
Speed 1000

Figure 13: Dataset Specific Animation Controller Containing all Sped Up Gesture Clips

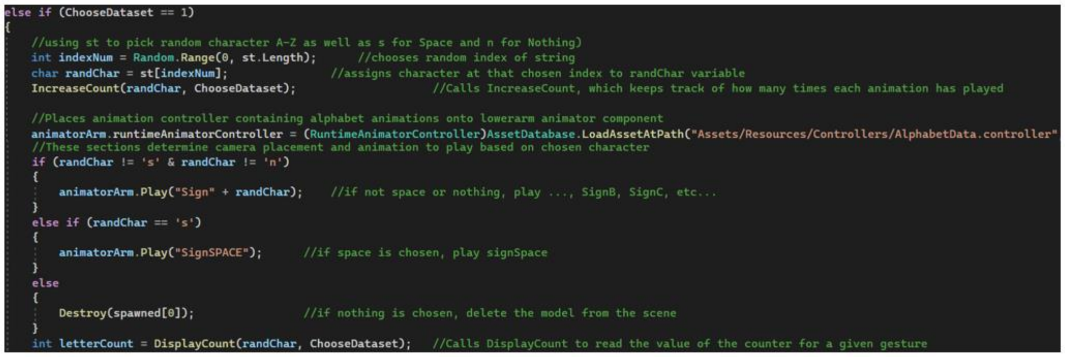
Once a MakeHuman model has been instantiated into the scene, the first step is to locate the lowerarm02.L bone as shown below so that an Animator component can be placed on it.



```
//Indexes through the models bone hierarchy until the lowerarm02 bone is reached
GameObject rig = model.transform.Find("MakeHuman default skeleton").gameObject;
GameObject root = rig.transform.GetChild(0).gameObject;
GameObject spine5 = root.transform.GetChild(2).gameObject;
GameObject spine4 = spine5.transform.GetChild(0).gameObject;
GameObject spine3 = spine4.transform.GetChild(0).gameObject;
GameObject spine2 = spine3.transform.GetChild(0).gameObject;
GameObject spine1 = spine2.transform.GetChild(2).gameObject;
GameObject clavicle = spine1.transform.GetChild(0).gameObject;
GameObject shoulder = clavicle.transform.GetChild(0).gameObject;
GameObject uparm1 = shoulder.transform.GetChild(0).gameObject;
GameObject uparm2 = uparm1.transform.GetChild(0).gameObject;
GameObject lowarm1 = uparm2.transform.GetChild(0).gameObject;
GameObject lowarm2 = lowarm1.transform.GetChild(0).gameObject;
```

Figure 14: Scripting Search for Lowerarm02 Bone of MakeHuman Model

Once located, it is possible to add the Animator component and enable it. Next, depending on user input of the dataset they wish to replicate, the correct Animation Controller is placed on the Animator. Playing an animation consists of selecting a random number/string index, where each number/character represents an animation. Depending on the character chosen, a distinct animation clip is played and a counter keeping track of the number of times each gesture has been performed is incremented.

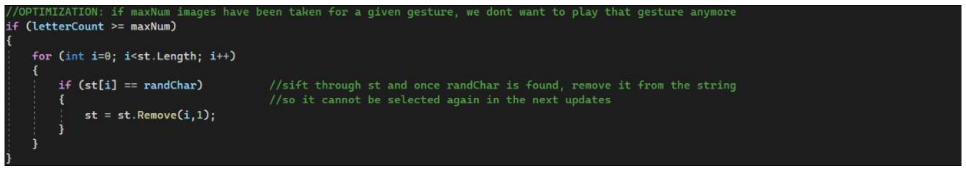


```
else if (ChooseDataset == 1)
{
    //using st to pick random character A-Z as well as s for Space and n for Nothing)
    int indexNum = Random.Range(0, st.Length);      //chooses random index of string
    char randChar = st[indexNum];                 //assigns character at that chosen index to randChar variable
    IncreaseCount(randChar, ChooseDataset);                         //Calls IncreaseCount, which keeps track of how many times each animation has played

    //Places animation controller containing alphabet animations onto lowerarm animator component
    animatorArm.runtimeAnimatorController = (RuntimeAnimatorController)AssetDatabase.LoadAssetAtPath("Assets/Resources/Controllers/AlphabetData.controller"
    //These sections determine camera placement and animation to play based on chosen character
    if (randChar != 's' & randChar != 'n')
    {
        animatorArm.Play("Sign" + randChar);    //if not space or nothing, play ..., SignB, SignC, etc...
    }
    else if (randChar == 's')
    {
        animatorArm.Play("SignSPACE");      //if space is chosen, play signSpace
    }
    else
    {
        Destroy(spawned[0]);                //if nothing is chosen, delete the model from the scene
    }
    int letterCount = DisplayCount(randChar, ChooseDataset);   //Calls DisplayCount to read the value of the counter for a given gesture
```

Figure 15: Alphabet Dataset Example for Randomly Selecting and Playing an Animation Clip

I also implemented an optimization for the time it would take to build the entire alphabet dataset (28 gestures total, 12,000 images per gesture) by removing the animation from the string of potential choices to be randomly chosen after it had been played the maximum number of times. Once all gestures have been performed the maximum number of times selected, the data capturing/collection subsystem stops the running of the Unity environment.



```
//OPTIMIZATION: if maxNum images have been taken for a given gesture, we dont want to play that gesture anymore
if (letterCount >= maxNum)
{
    for (int i=0; i<st.Length; i++)
    {
        if (st[i] == randChar)          //sift through st and once randChar is found, remove it from the string
        {                               //so it cannot be selected again in the next updates
            st = st.Remove(i,1);
        }
    }
}
```

Figure 16: Removal of Gesture Choice from String of Gestures

4.3. Subsystem Validation

The methodology of the model animation subsystem was validated manually during the validation process of the animation clips produced from the gesture data collection subsystem. I then implemented the process of adding components/controllers to sequentially spawned models through script, which was tested and validated on its own using an independent “ModelAnimation” script (which was then incorporated into the SceneController script in the completed Unity environment). Below you can observe the gesture animation K being performed on 3 unique, sequentially spawned MakeHuman models who did not have Animator components placed before being instantiated into the scene.

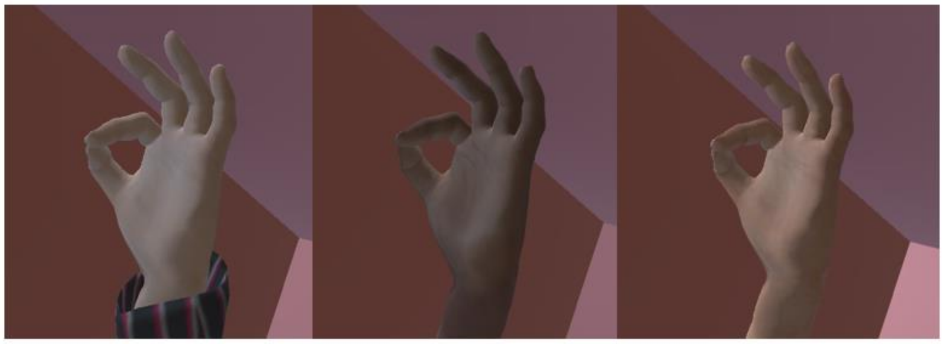
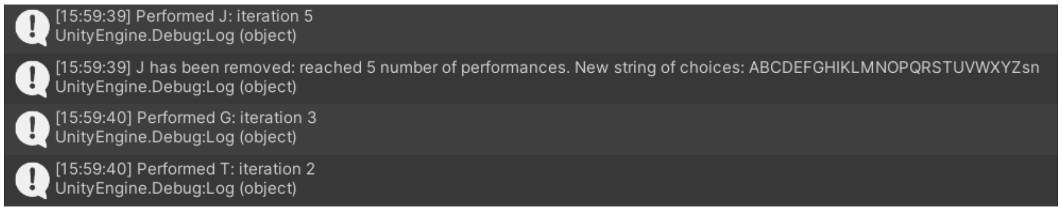


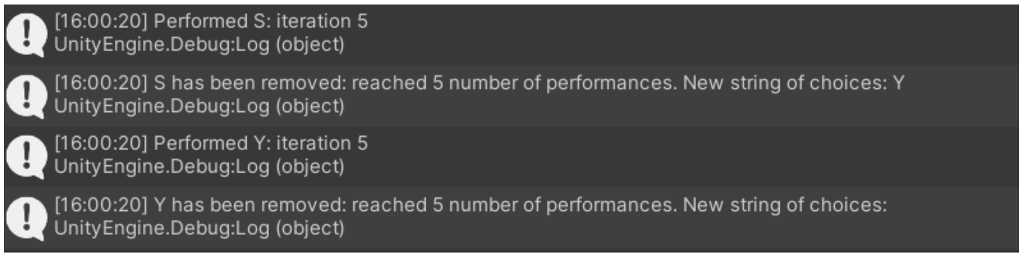
Figure 17: Gestures Performed on Distinct Imported MakeHuman Models Automatically

I validated that all animations from a given animation controller were able to be selected and played on any “Default” rigged MakeHuman model imported into the scene. I also validated that each animation was performed a maximum number of times selected by the user by checking the counter of each gesture by the end of the run. Finally, I validated that the subsystem would remove Alphabet animations from being selected to play after they had reached the maximum number of performances by running the system with a maximum number of 5 images per gesture and outputting to the console the number of times each gesture animation had been played. Once the console wrote that 5 images had been captured of a certain image, I printed the new string of choices and saw that the choice of the particular gesture had been removed.



```
[15:59:39] Performed J: iteration 5
UnityEngine.Debug:Log (object)
[15:59:39] J has been removed: reached 5 number of performances. New string of choices: ABCDEFGHIKLMNOPQRSTUVWXYZsn
UnityEngine.Debug:Log (object)
[15:59:40] Performed G: iteration 3
UnityEngine.Debug:Log (object)
[15:59:40] Performed T: iteration 2
UnityEngine.Debug:Log (object)
```

Figure 18: Log After First Letter J Reached Maximum Number of Iterations



```
[16:00:20] Performed S: iteration 5
UnityEngine.Debug:Log (object)
[16:00:20] S has been removed: reached 5 number of performances. New string of choices: Y
UnityEngine.Debug:Log (object)
[16:00:20] Performed Y: iteration 5
UnityEngine.Debug:Log (object)
[16:00:20] Y has been removed: reached 5 number of performances. New string of choices:
UnityEngine.Debug:Log (object)
```

Figure 19: Log After Last Letter Y Reached Maximum Number of Iterations

4.4. Subsystem Conclusion

The model animation subsystem has been fully validated and is functioning as designed. It uses the previously recorded animation clips from the gesture data collection subsystem and the sequentially spawned MakeHuman models from the human model generation subsystem to play each gesture animation a maximum user selected number of times. This proves that the created subsystems to this point are easily integrated together to form the full synthetic gesture training set generation system.

5. Data Capturing/Collection Subsystem Report

5.1. Subsystem Introduction

The data capturing/collection subsystem records images of the animated hand of each spawned model after it has performed a random gesture animation and stores the image as a .jpg in folders sorted by dataset and gesture using the scripted TakeImage method. This subsystem controls the placement of unique cameras in the scene using a PlaceCamera method to ensure that the proper camera angle is used to capture an image of the applied gesture. In addition, this subsystem introduces slight randomness in the position of the enabled camera to provide diverse data. Images are sized to be 512x512 pixels for neural network usage.

5.2. Subsystem Details

This subsystem works directly alongside the model animation subsystem, as both the PlaceCamera and TakeImage functions are called directly after a random animation is performed. In order for this subsystem to function properly, I had to create a number of virtual camera objects within the scene hierarchy that faced the performed gesture from the correct angle to ensure that the gesture is portrayed as it is meant to be. All cameras are disabled in the scene view at the beginning of the run.

The PlaceCamera script is responsible for enabling the correct camera depending on the gesture that was performed. For instance, gesture G requires a different camera than gesture B as shown in the figure below. This is because I was unable to record the gesture animation for G with my left hand rotated to the left using the LMC in a head mounted position. Instead, I recorded the gesture animation for G (and other letters) with my hand in the same orientation as I had with B, but altered the camera angle when looking at the played animation to make the gesture look as though it is horizontally oriented.

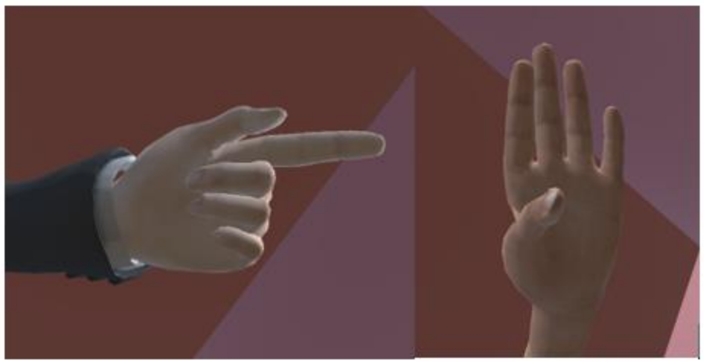
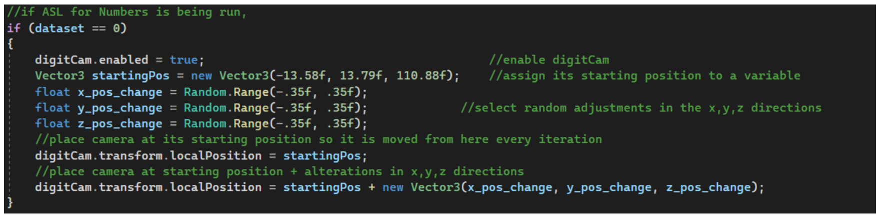


Figure 20: Camera Placement for Gesture G vs. B

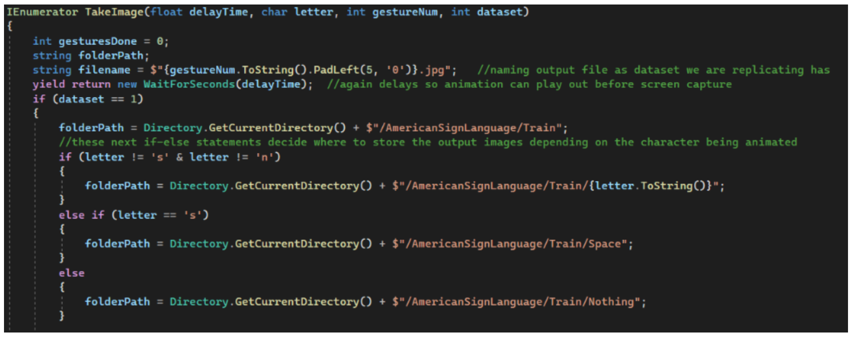
After the correct camera is enabled, a few random number generators select values within a predetermined range and add the random x,y, and z components to the coordinates of the starting camera position. By adding randomness in the camera position, we can ensure that the gesture will be viewed from a number of unique angles and add to the diversity of our data.



```
//if ASL for Numbers is being run,
if (dataset == 0)
{
    digitCam.enabled = true;                                //enable digitCam
    Vector3 startingPos = new Vector3(-13.58f, 13.79f, 110.88f);    //assign its starting position to a variable
    float x_pos_change = Random.Range(-.35f, .35f);
    float y_pos_change = Random.Range(-.35f, .35f);             //select random adjustments in the x,y,z directions
    float z_pos_change = Random.Range(-.35f, .35f);
    //place camera at its starting position so it is moved from here every iteration
    digitCam.transform.localPosition = startingPos;
    //place camera at starting position + alterations in x,y,z directions
    digitCam.transform.localPosition = startingPos + new Vector3(x_pos_change, y_pos_change, z_pos_change);
}
```

Figure 21: Scripting Randomness in Camera Position

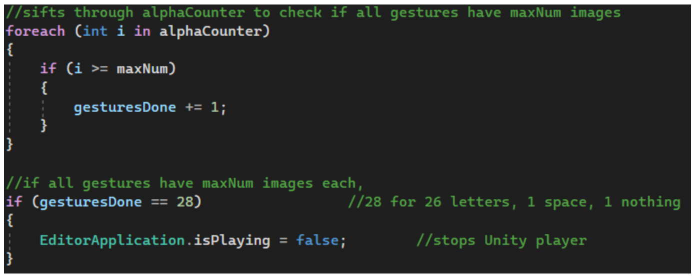
The TakeImage coroutine is responsible for recording a screenshot of the game view, naming the output file as well as the designating the output folder path depending on the chosen dataset and randomly played gesture animation. The Sign Language for Digits dataset named its files as the name of the gesture followed by the image number of that gesture, while the American Sign Language dataset only named images according to the image number. Depending on the dataset selected for replication using our system, the TakeImage coroutine first determines how to name the image following the same conventions mentioned above and determines where to store the gesture image depending on what gesture was signed.



```
IEnumerator TakeImage(float delayTime, char letter, int gestureNum, int dataset)
{
    int gesturesDone = 0;
    string folderPath;
    string filename = $"{gestureNum.ToString().PadLeft(5, '0')}.jpg";   //naming output file as dataset we are replicating has
    yield return new WaitForSeconds(delayTime);  //again delays so animation can play out before screen capture
    if (dataset == 1)
    {
        folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train";
        //these next if-else statements decide where to store the output images depending on the character being animated
        if (letter != 's' & letter != 'n')
        {
            folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train/{letter.ToString()}";
        }
        else if (letter == 's')
        {
            folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train/Space";
        }
        else
        {
            folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train/Nothing";
        }
```

Figure 22: TakeImage File Naming, Output Folder Designation, and Delay

Before taking the screenshot using the enabled camera from the PlaceCamera script, the TakeImage coroutine introduces a slight delay. The reason for this delay is because all animation clips begin in the same position, but over time move differently to form the final gesture sign. If a screenshot was recorded the instant that the gesture was performed, all images would resemble the sign five as that is the starting hand position of every created gesture clip. However, waiting for the gesture animation to play out in real time would take anywhere from 2-5 seconds depending on the complexity of the gesture. This would be hugely detrimental in the time it would take to create an entire dataset of 12,000 images per gesture. Instead, I sped up the animation clips by a factor of at 1000x and implemented a delay of .02 seconds so that I could ensure that each gesture image was taken with the final hand position in place. After the delay, the screenshot is taken and stored according to the settings discussed previously. Finally, the system checks the amount of screenshots that each gesture has stored and once all gestures have been recorded the maximum number of times, the script stops the Unity environment from running.



```
//sifts through alphaCounter to check if all gestures have maxNum images
foreach (int i in alphaCounter)
{
    if (i >= maxNum)
    {
        gesturesDone += 1;
    }
}

//if all gestures have maxNum images each,
if (gesturesDone == 28)                 //28 for 26 letters, 1 space, 1 nothing
{
    EditorApplication.isPlaying = false;        //stops Unity player
}
```

Figure 23: Alphabet Dataset Example of Dataset Completion/Environment Exit

5.3. Subsystem Validation

The validation of the data capturing/collection subsystem consisted of a number of tests. First, I ensured that the images were taken at the correct time after a gesture animation was performed, resulting in photos of the gestures looking as we intended (as explained in the previous section). I found that by delaying .02 units of time after the gesture was performed (while gestures were sped up 1000x), all images of each gesture were recorded after the entire animation played out and the hand was in its final position. This resulted in accurate gesture images and a decreased amount of time required to play/record all animations the maximum number of times.

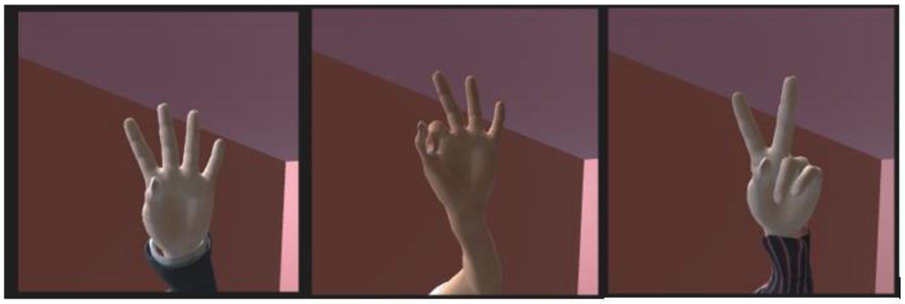


Figure 24: Example Validation for Images of Completed Gestures

I also validated this subsystem by ensuring that each gesture image was stored in the correct folder according to the dataset it belonged to and named following the same convention as the real dataset used. To test each of these, I ran the system using a maximum number of 5 images per gesture for both datasets. I allowed the system to play until it completed all required gestures and stopped itself. I then referred back to the folders I had created to store the output files and checked that each labeled folder contained images of the gesture that it specified (i.e. folder labeled one stored all images of one being portrayed). In addition, I validated that each image was named correctly (i.e. images of one being portrayed were named “one…”) and included a counter indicating the image number (up to 5 in this case).

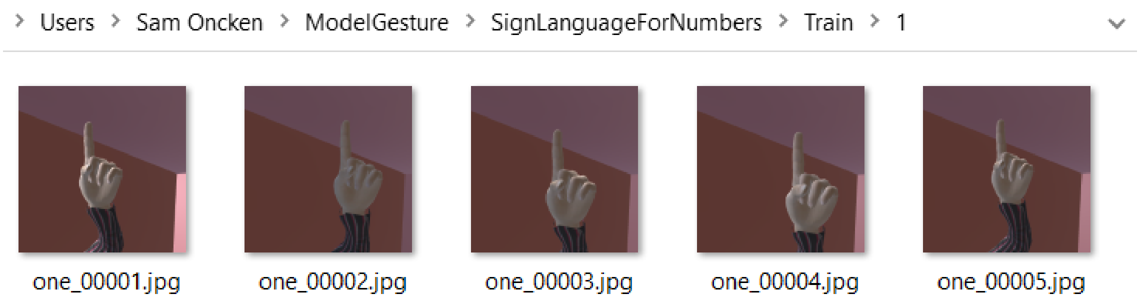

Users › Sam Oncken › ModelGesture › SignLanguageForNumbers › Train › 1
one_00001.jpg
one_00002.jpg
one_00003.jpg
one_00004.jpg
one_00005.jpg

Figure 25: Example Validation of Images Properly Named and Stored

Finally, I validated that the subsystem would stop the running of the Unity scene once all gesture clips were performed by running the scene and letting all animations play out until they reach the maximum number of performances and checking that the Unity environment will exit runtime after the condition is met.

5.4. Subsystem Conclusion

The data capturing/collection subsystem has passed all validation tests and is functioning as designed. This subsystem is able to record images directly after an animation has played on a randomly spawned model and store each into dataset specific and gesture specific folders as the real datasets we replicated had. In order for this subsystem to function properly, I had to use the implementations of the last 3 subsystems, proving that each subsystem can work together to form the virtual dataset generation environment. These subsystems form the backbone of the final system, but we will explore additional scene components/scripts that we implemented to improve our dataset generation system in the next section.

6. Dataset Generation Full System Report

6.1. System Introduction

The full dataset generation Unity environment is the combination of all previously described subsystems with the addition of a few extra customizations for increased diversity in data, which will be explained below. The dataset generation Unity environment is capable of fully replicating a real hand gesture dataset completely virtually. The system consists of one scene for recording gesture animations using the LMC and a MakeHuman model and another scene for importing random models, applying gesture animations, and recording/storing quality images of each hand gesture to form an entire virtual dataset. We have completed virtual dataset replication of an American Sign Language letters dataset (28 gestures) and a Sign Language for Numbers dataset (11 gestures) consisting of 12,000 images per gesture and we are currently working on implementing a replication of the HaGRID dataset, which includes full body images (useful continuation of our research into winter break and 404).

6.2. System Details

As described in the data capturing/collection subsystem report, we have validated that all subsystems can work together to form a consistent dataset generation environment. All scripting for each of the subsystems including the placement of animation components, playing of random animation clips depending on the dataset selected, placement of virtual cameras, recording/storing of images, and conditional logic to quit the Unity run once the dataset is fully generated have been merged into a single script called SceneController.cs. From the SceneController script, users can select the dataset they wish to replicate along with the amount of images per gesture they would like to have outputted and with the click of a play button, the dataset generation process will begin.

To provide a brief overview of the dataset generation system process, the first step involves the human model generation subsystem, which imports all rigged MakeHuman models from the assets folder into an array of strings by name. Every 30 frames, a random index of this array is chosen and the name of the model corresponding to this index is instantiated into the scene view of the environment. Next, the model animation subsystem is called, where depending on the dataset chosen to be replicated, a random animation is applied to the model. Next, the data capturing/collection subsystem is called where (depending on the performed animation) the correct camera is enabled in the scene, the output file is named and the folder path is selected, and a screenshot is captured and stored. This process repeats until all animations have been played a maximum number of times and the Unity environment stops running. At this point, the finalized dataset is created.

6.2.1. Additional System Characteristics

6.2.1.1. Randomization in Background Conditions

As instructed by our graduate student sponsor, we have additionally incorporated a background image randomization script to increase diversity in our virtual datasets. This script was written by our graduate student sponsor Pranav Dhulipala, but he gave us permission to use it within our scene. In order to incorporate this script, I created wall objects around the location where the models would be spawned. By simply placing the wall objects into the public variable Walls array within the background randomization script, upon running the environment, each wall will appear as a random image from the Describable Textures Dataset[3] found online and linked above.

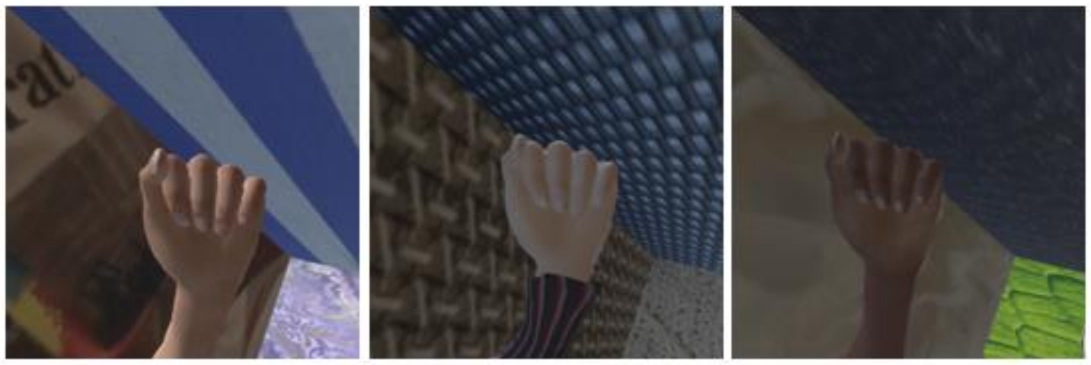


Figure 26: Background Image Randomization within Images of Gesture A

6.2.1.2. Randomization in Lighting Conditions

The final environment also includes a lighting condition randomization feature, where the intensity of each of the two light sources within the scene hierarchy are randomly set after each iteration of a gesture animation playing. We are attempting to replicate real world scenarios as best as possible, where lighting conditions are not always optimal and hands might appear darker or brighter than wanted. We plan for a gesture recognition neural network trained using our synthetic data to recognize such outliers.

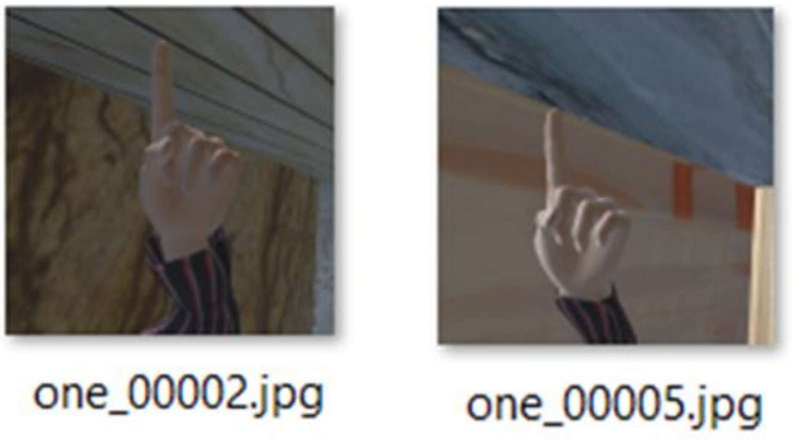


Figure 27: Alterations in Lighting from Image 2 vs. Image 5 on Gesture 1

6.3. System Validation

The final system was validated first by running the Unity environment under various user-selected conditions. I tested the Unity dataset generation system using the Sign Language for Numbers dataset and selection of 5, 10, 50, 1,500, and 12,000 images per gesture. After each run, I validated that the gesture images were accurately recorded and stored with correct naming conventions. I validated that each picture clearly contained the hand gesture without too much cut out of frame as well. Next, I tested the same things using the American Sign Language dataset replication choice selected. All of these tests passed as all images were accurately taken of each gesture and stored in their corresponding folder paths. The 12,000 image per gesture dataset is the final form of the replicated synthetic training set used for testing.

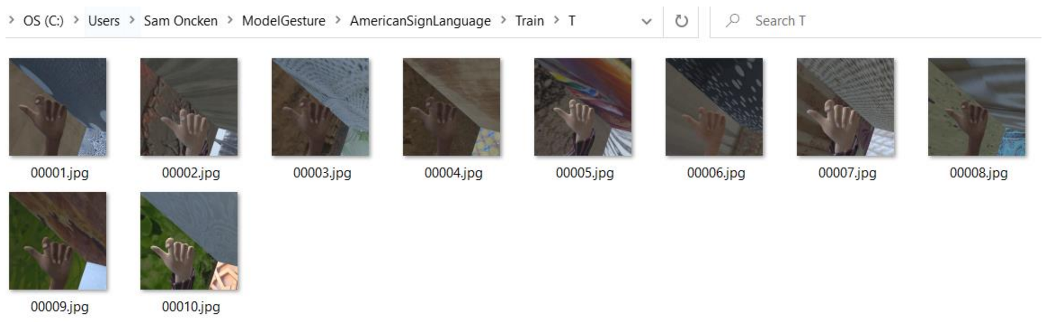


Figure 28: Validation Example - Letter T Performed and Stored 10 Times

I also validated the scene customizations by running the RandomTexture (random background image script) alone to see whether each wall changed appearance. While running both scripts at the same time, it was determined that the best speed at which we should change scene background was .12 units of time, which allowed for a new background set to appear after each new iteration of an animation playing.

Finally, I validated the light source randomization script by inspecting each of the output images of a certain gesture class to make sure that images were not all subject to the same light source intensity as shown below.

6.4. System Conclusion

The full system passed all validation tests, proving that all previously designed subsystems were integrated successfully. The final dataset generation system has been completed. Using the finalized system, we were able to replicate both the Sign Language for Numbers dataset as well as the American Sign Language dataset which contained 132,000 and 336,000 images respectively. These dataset outputs are stored on the lab computer in WEB 156 and are currently being pipelined into the neural networks that Steven is training and testing with, which will be described in the next section.

7. Virtual Training Set Testing Report

7.1. Training Set Testing Introduction

The training set testing involves training image classification Convolution Neural Networks (CNNs) using the benchmark datasets we found or the synthetic datasets we created and testing their hand gesture recognition accuracy. The CNN used so far is ResNet18, a well known image classification model. All training and testing is done using the Keras API in TensorFlow with Python. After a model is trained with a certain dataset, tests are run to see the model's ability to classify new images it hasn't seen before that are from the same dataset. This is done with both the real benchmark dataset and our synthetic dataset, and the accuracy metrics are compared to see if synthetic data can result in similar accuracy results when testing the model. Then, further validation for training on synthetic datasets and testing on real data is done to prove the ability to use CNNs trained on synthetic data for hand gesture recognition with real data. This opens up the possibility to use synthetic data to augment or potentially even replace real data when training CNN models.

7.2. Training Set Testing Progress/Methodology

The process starts with creating an input pipeline for the datasets using built in functions within the Keras API of TensorFlow. The input pipeline allows for pulling of images in batches from specified dataset directories instead of loading the entire dataset into an array, which prevents the system from running out of memory. Preprocessing is applied in the pipeline, where we normalized the values of the pixels to be values from zero to one, and randomly flipped the images horizontally to simulate both left and right hands since our data had only left handed images. The batches of images are also split into a 50/25/25 training, validation, and testing split. Three pipelines are created for the benchmark dataset, our synthetic dataset, and our personally created dataset. The next step involves setting up the model for training and testing. The ResNet18 model is created using an image classifiers Python package. Adjustments are made to the base model, either adding more convolution layers, and/or freezing the weights for transfer learning. All models must have a final global average pooling layer and a densely connected layer that has the same number of neurons as there are classes in the dataset for the final classification.

With the setup complete, the model is then compiled to specify the loss, accuracy, and optimizer used during training. Training is run using one of the datasets, and the metrics are recorded and graphed. A test using the testing partition of the training dataset is then run to ensure the model's ability to classify images from the training dataset. The model is then tested on one of the other datasets to validate its ability to generalize when classifying images that differ from the training dataset. The accuracy of the test is recorded, and a confusion matrix is generated to visualize the classification of the test images.

The following results are from the American Sign Language digits dataset. Due to constraints in drive space, we could not run tests on the other datasets since they were too large. However, we now have the capability to run these tests on the other datasets and plan on doing so over Winter break. The three datasets used are the synthetic dataset, the real benchmark dataset, and a real dataset we made ourselves consisting of images of my own hands.

For the synthetic training, tests were run using transfer learning and training all weights from scratch. However, when using pre-trained imagenet weights, the model showed almost no improvements in training accuracy and virtually no improvement in validation accuracy. After adding two trainable convolution layers with 512 filters, the training accuracy improved to around 50% after 30 epochs, but validation accuracy still had no improvements at all. Even after adding two more convolution layers with double the filters, the training accuracy reached around 80% and validation accuracy still had no improvements.

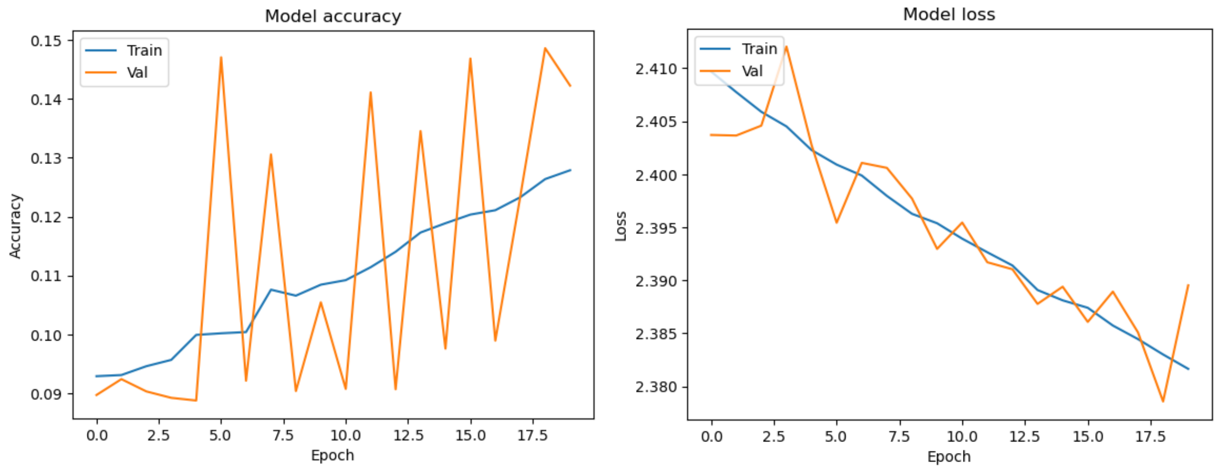
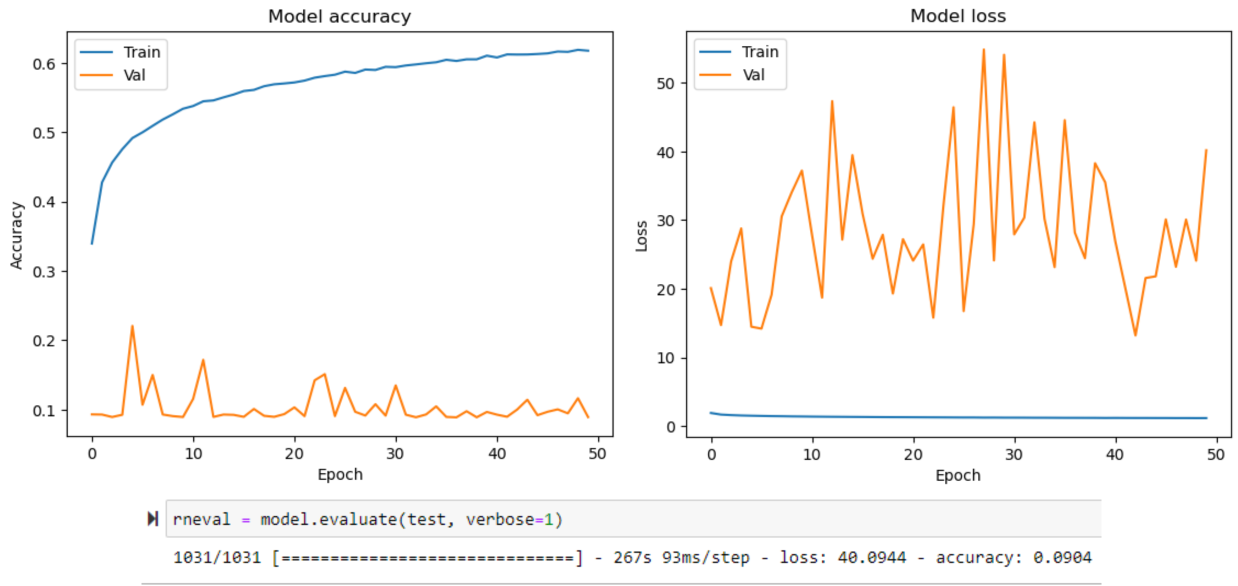


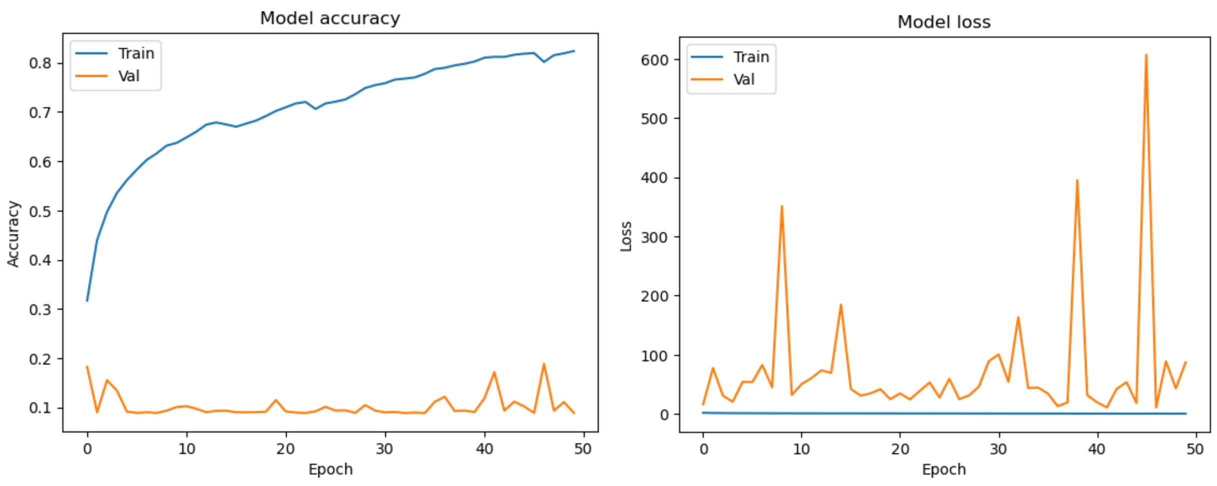
Figure 29: Synthetic Data Training and Evaluation with no Additional Layers Using Transfer Learning

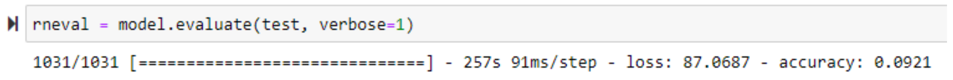


```
rneval = model.evaluate(test, verbose=1)

1031/1031 [==============================] - 267s 93ms/step - loss: 40.0944 - accuracy: 0.0904
```

Figure 30: Synthetic data Training and Evaluation with 2 Additional Convolution Layers (2x512) Using Transfer Learning



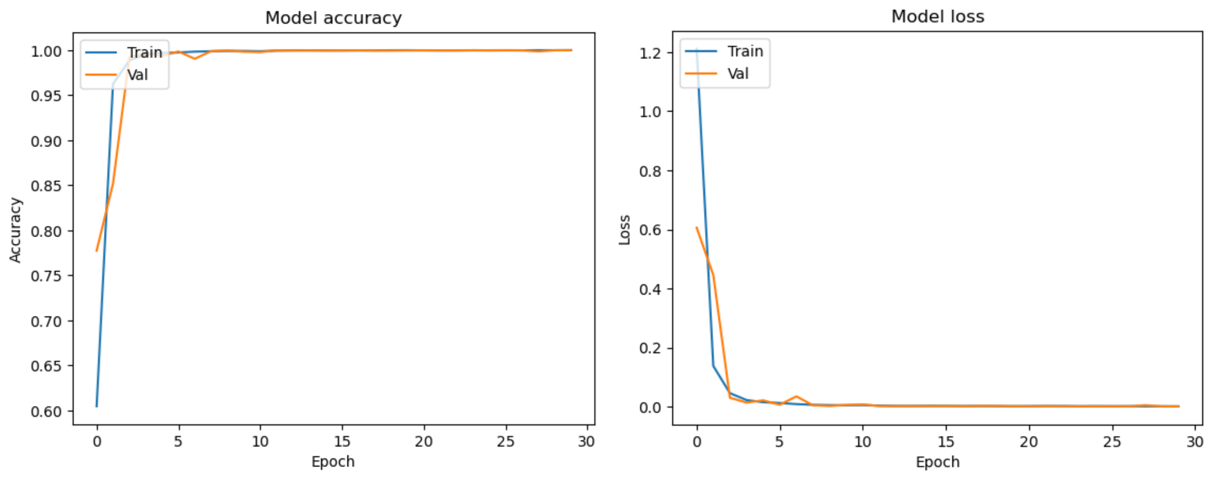


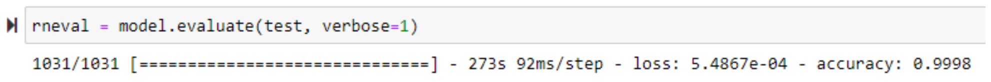
```
rneval = model.evaluate(test, verbose=1)

1031/1031 [==============================] - 257s 91ms/step - loss: 87.0687 - accuracy: 0.0921
```

Figure 31: Synthetic Data Training and Evaluation with 4 Additional Convolution Layers (2x512, 2x1024) Using Transfer Learning

With training from scratch with the synthetic data and no extra convolution layers, the training and validation accuracy reached 100% after 30 epochs of training. The testing accuracy on the test partition was 99.98%. However, when tested on the real benchmark data, the accuracy was only 7.24%, and the confusion matrix showed that the model was completely misclassifying the images. This was also seen when testing on our own created dataset, which had 19.73% accuracy testing, and was also clearly misclassifying images when viewing the confusion matrix.

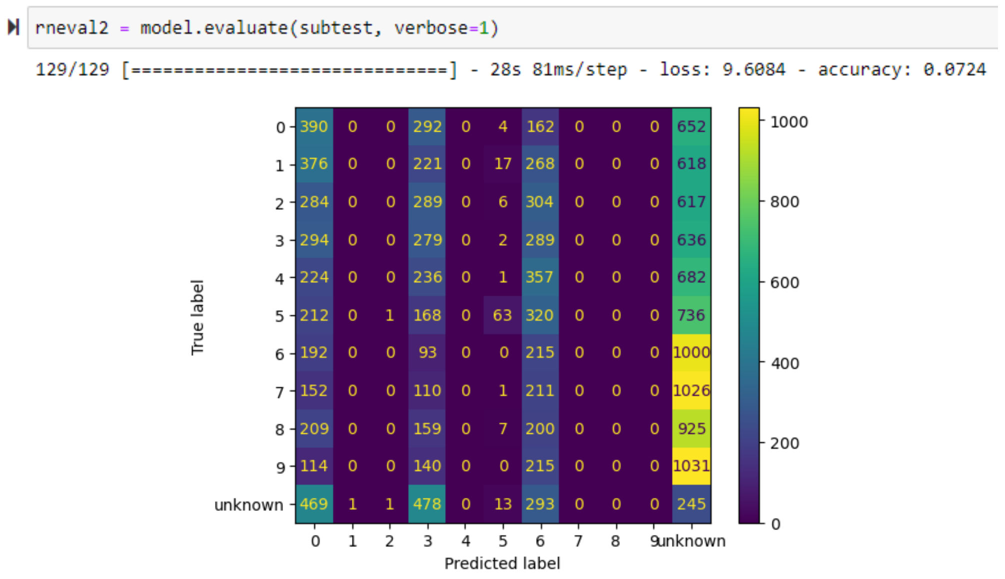




```
rneval = model.evaluate(test, verbose=1)

1031/1031 [==============================] - 273s 92ms/step - loss: 5.4867e-04 - accuracy: 0.9998
```

Figure 32: Synthetic data Training and Evaluation with no Additional Layers, Training Weights from Scratch

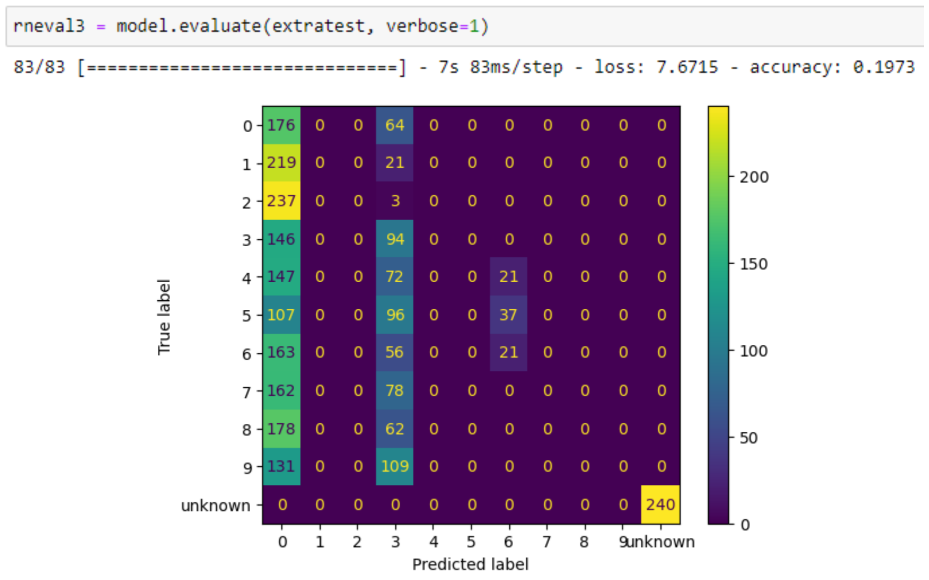


```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 28s 81ms/step - loss: 9.6084 - accuracy: 0.0724
```

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	unknown
0	390	0	0	292	0	4	162	0	0	0	652
1	376	0	0	221	0	17	268	0	0	0	618
2	284	0	0	289	0	6	304	0	0	0	617
3	294	0	0	279	0	2	289	0	0	0	636
4	224	0	0	236	0	1	357	0	0	0	682
5	212	0	1	168	0	63	320	0	0	0	736
6	192	0	0	93	0	0	215	0	0	0	1000
7	152	0	0	110	0	1	211	0	0	0	1026
8	209	0	0	159	0	7	200	0	0	0	925
9	114	0	0	140	0	0	215	0	0	0	1031
unknown	469	1	1	478	0	13	293	0	0	0	245

Figure 33: Evaluation and Confusion Matrix from Test Using Real Benchmark Dataset



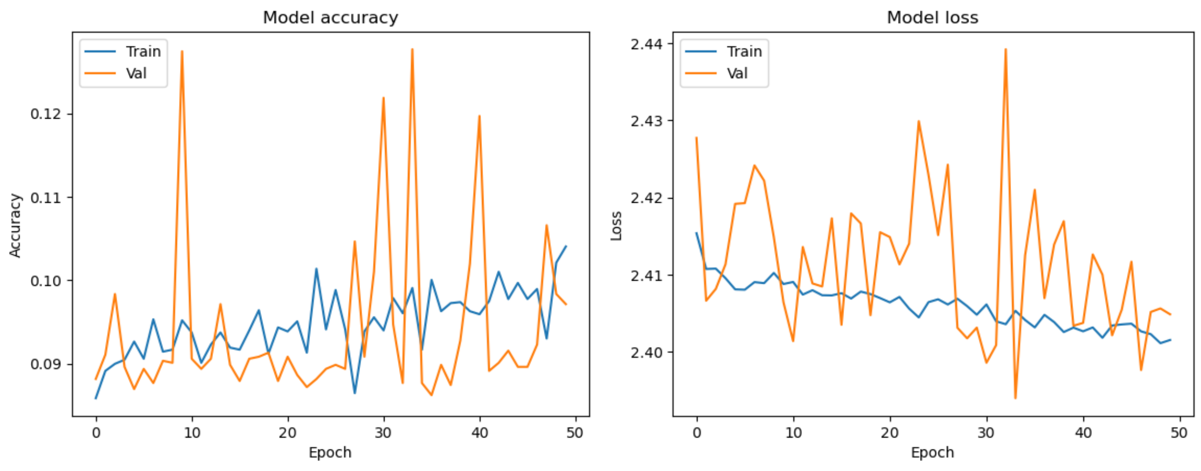
```
rneval3 = model.evaluate(extratest, verbose=1)
```

```
83/83 [==============================] - 7s 83ms/step - loss: 7.6715 - accuracy: 0.1973
```

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	unknown
0	176	0	0	64	0	0	0	0	0	0	0
1	219	0	0	21	0	0	0	0	0	0	0
2	237	0	0	3	0	0	0	0	0	0	0
3	146	0	0	94	0	0	0	0	0	0	0
4	147	0	0	72	0	0	21	0	0	0	0
5	107	0	0	96	0	0	37	0	0	0	0
6	163	0	0	56	0	0	21	0	0	0	0
7	162	0	0	78	0	0	0	0	0	0	0
8	178	0	0	62	0	0	0	0	0	0	0
9	131	0	0	109	0	0	0	0	0	0	0
unknown	0	0	0	0	0	0	0	0	0	0	240

Figure 34: Evaluation and Confusion Matrix from Test Using my Hands Dataset

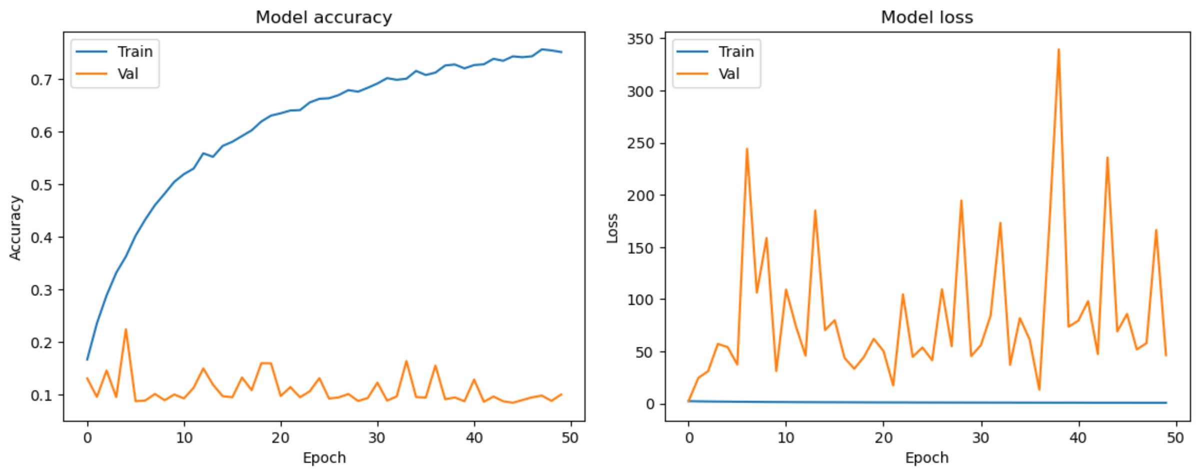
Using transfer learning training with the real benchmark data resulting in similar results as with the synthetic data. Training with no extra convolution layers resulted in almost no improvements in training and validation accuracy after 50 epochs. After adding four convolution layers (the first two with 512 filters and the last two with 1024 filters), the training accuracy improved to around 75%, while the validation accuracy showed no improvements again.



```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 35s 90ms/step - loss: 2.4065 - accuracy: 0.1028
```

Figure 35: Benchmark Dataset Training and Evaluation with no Additional Layers Transfer Learning

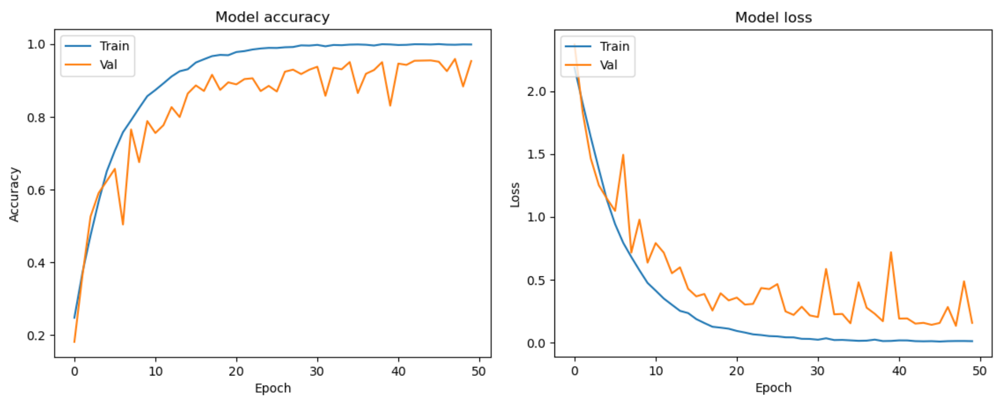


```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 25s 79ms/step - loss: 46.5351 - accuracy: 0.0984
```

Figure 36: Benchmark Dataset Training and Evaluation with Added Convolution Layers (2x512, 2x1024) Transfer Learning

When training using the real benchmark dataset from scratch with no additional convolution layers, 50 epochs achieved 99.84% and 95.3% accuracy for training and validation accuracy respectively. When tested on the synthetic data, the accuracy was only 9.31%, and on our real data it was 22.05%, with both confusion matrices showing clear misclassification of the data.



```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 25s 76ms/step - loss: 0.1566 - accuracy: 0.9529
```

Figure 37: Benchmark Dataset Training and Evaluation with no Additional Layers, Training from Scratch

```
rneval = model.evaluate(test, verbose=1)
```

```
4125/4125 [==============================] - 439s 106ms/step - loss: 9.1008 - accuracy: 0.0931
```

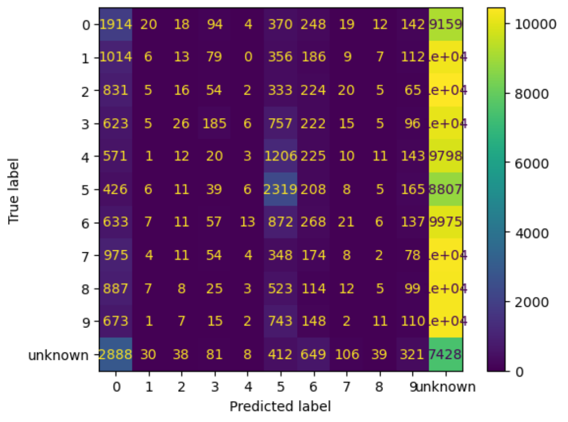


Figure 38: Evaluation and Confusion Matrix from Test using Synthetic Dataset

```
rneval3 = model.evaluate(extratest, verbose=1)
```

```
83/83 [==============================] - 7s 83ms/step - loss: 4.0121 - accuracy: 0.2205
```

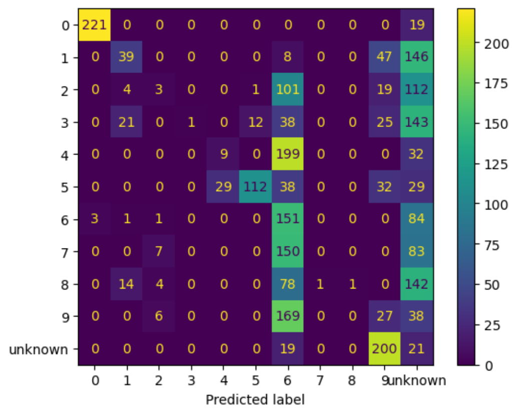


Figure 39: Evaluation and Confusion Matrix from Test Using my Hands Dataset

Below are two tables representing the data from above. For the transfer learning training, we did not run the tests (represented as 'X') because the model was already unable to validate correctly on the training data and would show no better than a guess for the image classification. For the tests from scratch, as accuracy roughly matched the training accuracies >95%, the latter tests were then run.

The following table shows data from synthetic training and testing after 50 epochs for transfer learning and after 30 epochs for learning from scratch.

Table 1: Synthetic Data Training and Testing

Additional Changes	Training Method	Final Training Accuracy (%)	Final Validation Accuracy (%)	Final Evaluation Accuracy (%)	Benchmark Data Test Accuracy (%)	My Hands Data Test Accuracy (%)
No added layers	Transfer learning	12.79	14.23	X	X	X
	Scratch	100	100	99.98	7.24	19.73
2 convolution layers (2x512)	Transfer learning	61.77	8.91	X	X	X
4 convolution layers (2x512, 2x1024)	Transfer learning	82.38	8.88	X	X	X

The following table shows data from real benchmark training and testing after 50 epochs for both transfer learning and from scratch.

Table 2: Real Benchmark Data Training and Testing

Additional Changes	Training Method	Final Training Accuracy (%)	Final Validation Accuracy (%)	Final Evaluation Accuracy (%)	Synthetic Data Test Accuracy (%)	My Hands Data Test Accuracy (%)
No added layers	Transfer learning	10.40	9.71	10.28	X	X

	Scratch	99.84	95.3	95.29	9.31	22.05
4 convolution layers (2x512, 2x1024)	Transfer learning	75.07	10.08	9.84	X	X

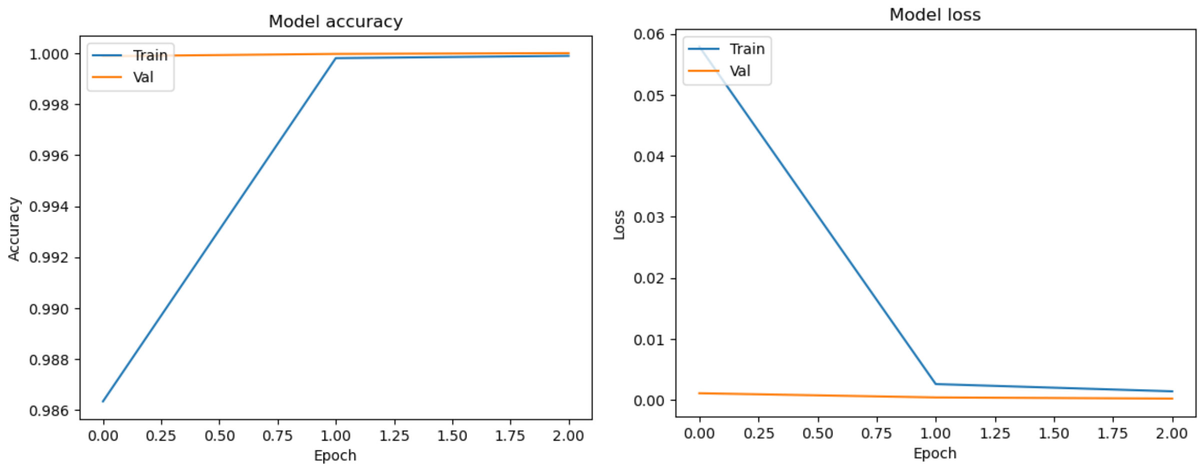
7.3. Training Set Testing Problem Diagnostic and Mitigation

When training from scratch on the real benchmark ASL digits dataset, the model's failure to classify images is likely due to the small size of the dataset. There are only 1,500 images per gesture with 11 gestures, totaling 16,500 images. For image classification, this is far too small to properly train any type of significant models for classification. The data is also black and white, which gives the models less information to work with as more data leads to a more robust and accurate model.

When training from scratch on the synthetic ASL digits dataset, the reality gap between synthetic data and real data is most likely the source of the model's failure to classify the gestures accurately. As the dataset has 12,000 images per gesture totaling 132,000 images, the size could be larger, but it is large enough to achieve good accuracy for classification. Knowing this, the model would need to be fed data that has more diversity of information, especially considering the background since our synthetic data's background is not necessarily representative of realistic backgrounds.

With transfer learning on both the synthetic and real ASL digits datasets, the model wasn't able to improve in validation accuracy in either case. There are many possibilities for why this is happening, but what we believe to be the issue is the difference between the imagenet data and our data. Imagenet generally contains objects and animals, and as we are doing hand gesture recognition, we believe the features from the convolution layers using imagenet weights are causing the model to fail. Even after adding convolution layers to the end of the ResNet18 layers that were trainable, the training accuracy slowly increased while the validation accuracy still showed no improvements.

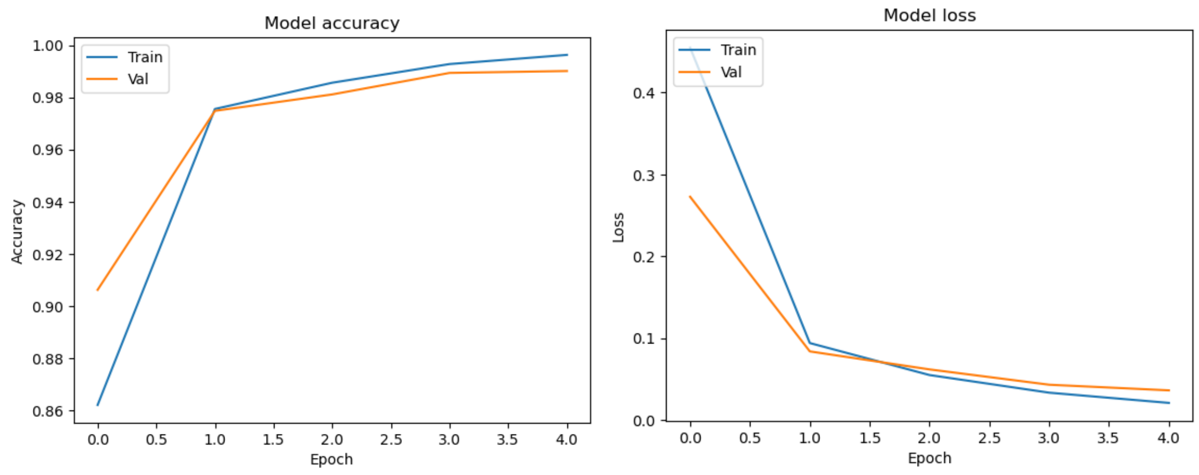
Training was also done using the imagenet weights but leaving them trainable, and then training with the synthetic data. The finished model was then immediately trained again on the real data, and tested on our real dataset. After speaking to our sponsor, they said retraining the imagenet weights is something we should not do. Though the method was slightly wrong, the results are reported below.



```
rneval = model.evaluate(test, verbose=1)
```

```
1031/1031 [==============================] - 273s 98ms/step - loss: 2.2976e-04 - accuracy: 1.0000
```

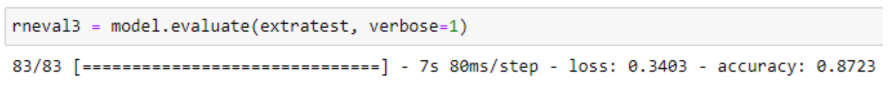
Figure 40: Synthetic Training and Evaluation Using Trainable Imagenet Weights



```
rneval2 = model.evaluate(subtest, verbose=1)
```

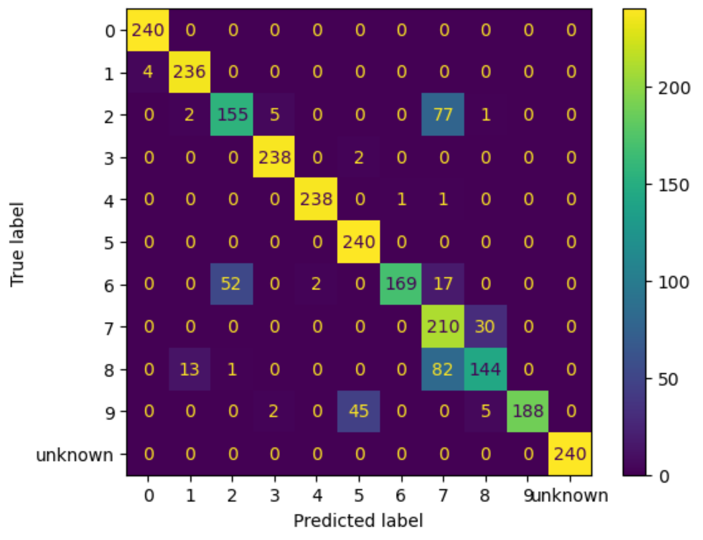
```
129/129 [==============================] - 25s 76ms/step - loss: 0.0333 - accuracy: 0.9891
```

Figure 41: Real Training and Evaluation using Trainable Imagenet Weights after the Synthetic Training



```
rneval3 = model.evaluate(extratest, verbose=1)

83/83 [==============================] - 7s 80ms/step - loss: 0.3403 - accuracy: 0.8723
```



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	unknown
0	240	0	0	0	0	0	0	0	0	0	0
1	4	236	0	0	0	0	0	0	0	0	0
2	0	2	155	5	0	0	0	77	1	0	0
3	0	0	0	238	0	2	0	0	0	0	0
4	0	0	0	0	238	0	1	1	0	0	0
5	0	0	0	0	0	240	0	0	0	0	0
6	0	0	52	0	2	0	169	17	0	0	0
7	0	0	0	0	0	0	0	210	30	0	0
8	0	13	1	0	0	0	0	82	144	0	0
9	0	0	0	2	0	45	0	0	5	188	0
unknown	0	0	0	0	0	0	0	0	0	0	240

Figure 42: Test Evaluation and Confusion Matrix after Synthetic and Real Training using Trainable Imagenet Weights and My Hands Dataset

The following table shows the synthetic and real benchmark training and testing. The order in this table is the order of training, with synthetic first and real data following. The synthetic data was trained for 3 epochs and the real data was trained for 5 epochs. After all the training, the test on our dataset was run. The test with our data was run after fully training with both datasets, so the test accuracy with our data for the synthetic training was not run (represented with ‘X’).

Table 3: Synthetic and Real Benchmark Training and Testing

Training Data	Training Method	Final Training Accuracy (%)	Final Validation Accuracy (%)	Final Evaluation Accuracy (%)	My Hands Data Test Accuracy (%)
Synthetic	Trainable imagenet weights	99.99	100	100	X
Real benchmark	Trainable imagenet weights	99.62	99.01	98.91	87.23

As seen in the table and data above, the model is able to classify the images correctly with only some mistakes being made for certain gestures. This accuracy for testing on our dataset of 87.23% was significantly higher than the 19.73% test accuracy by training the model from scratch with synthetic data and 22.05% test accuracy by training the model from scratch with the real benchmark data. Though the method was wrong, this method did reveal the direction our training should go, which is augmenting the real data with synthetic data. This would make up for the reality gap with the synthetic data and the lack of images with the real data.

Going forward into Winter break, we will need to continue with training using both synthetic and real data using correct training methods and different model structures to optimize and improve the test accuracy. We will also need to expand our dataset to have more images per gesture and have more variances in backgrounds and lighting to truly test the robustness of our trained models. We will continue to run these tests on the ASL digits dataset, as well as the ASL alphabet dataset.

7.4. Training Set Testing Conclusion

Though our testing of datasets was delayed and the initial testing showed accuracies well below acceptable values, we are moving in the right direction with our testing and will make progress over Winter break towards improving the test accuracy through correct model training methods. True transfer learning with ResNet18 showed no improvements in accuracy results, so it is likely that we will continue with training from scratch with no pre-trained weights. We will also continue using structures of well known image classification models, potentially expanding to models other than ResNet18. Once the adjustments are made to the training, we will be able to take large datasets and train image classification CNNs to recognize hand gestures accurately.