Hand Gesture Recognition

Samuel Oncken, Steven Claypool

FINAL REPORT

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

CONCEPT OF OPERATIONS

REVISION - 4
21 April 2023

CONCEPT OF OPERATIONS
FOR
Hand Gesture Recognition

TEAM <72>

APPROVED BY:

Samuel Oncken	4/21/2023
_____	
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
4	4/21/2023	Samuel Oncken Steven Claypool		Revision for Final 404 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 machine learning and artificial intelligence have been written in the last decade, proving its increasing importance across various industries in society. Our research focuses on a distinct category of machine learning and AI: computer vision. More specifically, we seek to uncover how well virtual (synthetic) data can train a neural network to recognize hand gestures. The first stage of this project is to create a user-friendly Unity environment where a dataset organizer can produce comprehensive and scalable hand gesture datasets capable of training a hand gesture recognition neural network. The next stage of this project involves understanding and training existing image classification Convolutional Neural Networks (CNNs) to compare the recognition accuracy when training with various compositions of real and virtual data [1,2,3]. Additionally, a faster RCNN will be used to analyze the usefulness of synthetically generated data in terms of object detection/localization in the Full System Report within this document.

Many of the datasets we have found online, summarized by [4], 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 hiring several different humans to perform 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 [5] 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, therefore it is required that a machine learning model be trained on hand gesture data recorded from many different perspectives.

Our research is aimed at bringing these important considerations together in the generation of a "virtual" dataset. In a virtual dataset, instead of having several human subjects record their movements manually, we will be recording the movements of one human (under various conditions such as lighting, distance from sensor, etc.) and applying the motion to randomly generated 3-D virtual human models within the Unity game engine environment. In addition, within Unity, every iteration of a gesture performance will be recorded using multiple virtual cameras to improve gesture recognition from numerous angles.

We have uncovered a handful of research papers pursuing the application of synthetic data in computer vision algorithms from a wide variety of topics including but not limited to satellite imagery [6], robotics [7], and random object detection [8]. Similar research has been done by a Texas A&M alumnus in which a virtual training set of hand gestures was used to increase the recognition accuracy of gesture recognition neural networks [9].

We are looking to expand upon this student's research because we believe that this method can prove to be a cost-effective solution that can produce high recognition accuracy results, which we will be evaluating when testing our virtual datasets within hand gesture recognition neural networks in a future portion of this document.

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. We shall also generate a full body dataset to use in the faster RCNN object detection analysis.

2.3. Referenced Documents and Standards

[1] J. Nagi et al., "Max-pooling Convolutional Neural Networks for vision-based hand gesture recognition," IEEE Xplore, 2011. [Online]. Available: https://ieeexplore.ieee.org/document/6144164/. [Accessed: 5-Sep-2022].

[2] J. Yu, M. Qin, and S. Zhou, "Dynamic gesture recognition based on 2D convolutional neural network and feature fusion," Research Gate, Mar-2022. [Online]. Available: https://www.researchgate.net/publication/359223015_Dynamic_gesture_recognition_based_on_2D_convolutional_neural_network_and_feature_fusion. [Accessed: 5-Sep-2022].

[3] Karen Simonyan, Andrew Zisserman, "Very Deep Convolution Networks for Large-Scale Image Recognition," arXiv:1409.1556, 04-Sep-2014. [Online]. Available:

https://arxiv.org/abs/1409.1556. [Accessed: 20-Dec-2022].

[4] M. Asadi-Aghbolaghi et al., “A Survey on Deep Learning Based Approaches for Action and Gesture Recognition in Image Sequences,” IEEE Xplore, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7961779. [Accessed: 4-Sep-2022].

[5] C. C. de Amorim and C. Zanchettin, “ASL-Skeleton3D and ASL-Phono: Two Novel Datasets for the American Sign Language,” arXiv:2201.02065, 06-Jan-2022. [Online]. Available: https://arxiv.org/abs/2201.02065. [Accessed: 5-Sep-2022].

[6] J. Shermeyer, T. Hossler, A. Van Etten, D. Hogan, R. Lewis, and D. Kim, “RarePlanes: Synthetic Data Takes flight,” arXiv:2006.02963, 10-Nov-2020. [Online]. Available: https://arxiv.org/abs/2006.02963. [Accessed: 3-Oct-2022].

[7] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, “Sim-to-real via sim-to-SIM: Data-efficient robotic grasping via randomized-to-canonical adaptation networks,” arXiv:1812.07252, 21-Jul-2019. [Online]. Available: https://arxiv.org/abs/1812.07252. [Accessed: 3-Oct-2022].

[8] S. Hinterstoisser, O. Pauly, H. Heibel, M. Marek, and M. Bokeloh, “An Annotation Saved is an Annotation Earned: Using Fully Synthetic Training for Object Instance Detection,” arXiv:1902.09967, 26-Feb-2019. [Online]. Available: https://arxiv.org/abs/1902.09967. [Accessed: 3-Oct-2022].

[9] Z. Helton, “VR Hand Tracking for Robotics Applications”, unpublished.

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 shall be composed of images of the hand gestures performed on synthetic human models. 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 and is implemented in the HANDSv2 dataset described in a future portion of this report.

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.

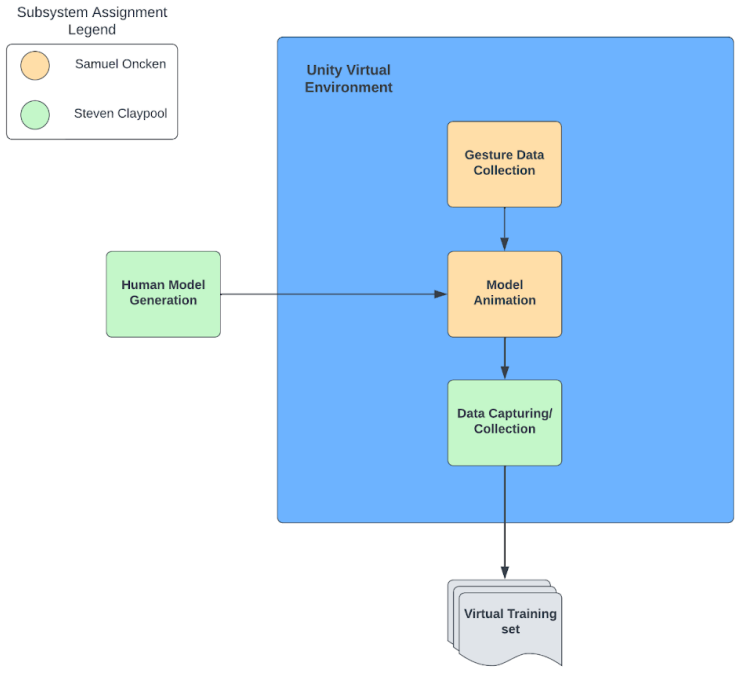


Figure 1: Flowchart of Virtual Hand Gesture Recognition Training Set Generation System

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 - 3
21 April 2023

FUNCTIONAL SYSTEM REQUIREMENTS
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72 4/21/2023
Author Date

APPROVED BY:

Samuel Oncken 4/21/2023
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
3	4/21/2023	Samuel Oncken Steven Claypool		Release of 404 Final Report

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.

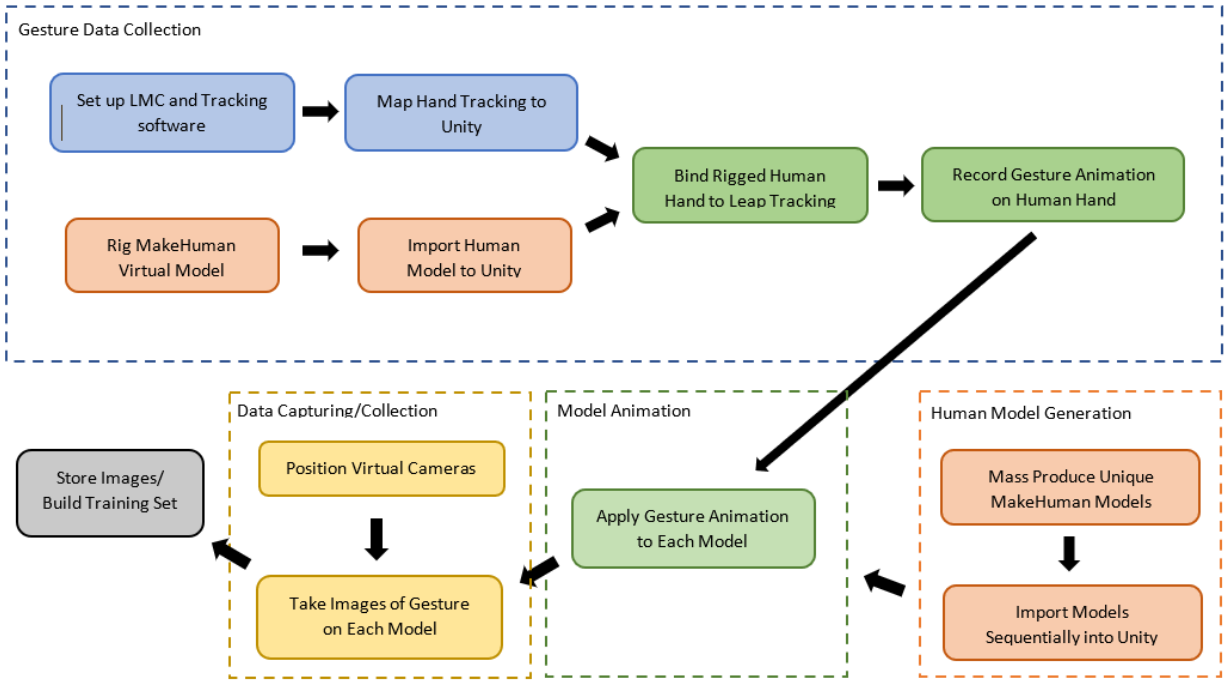


Figure 1. Project Conceptual Image

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.

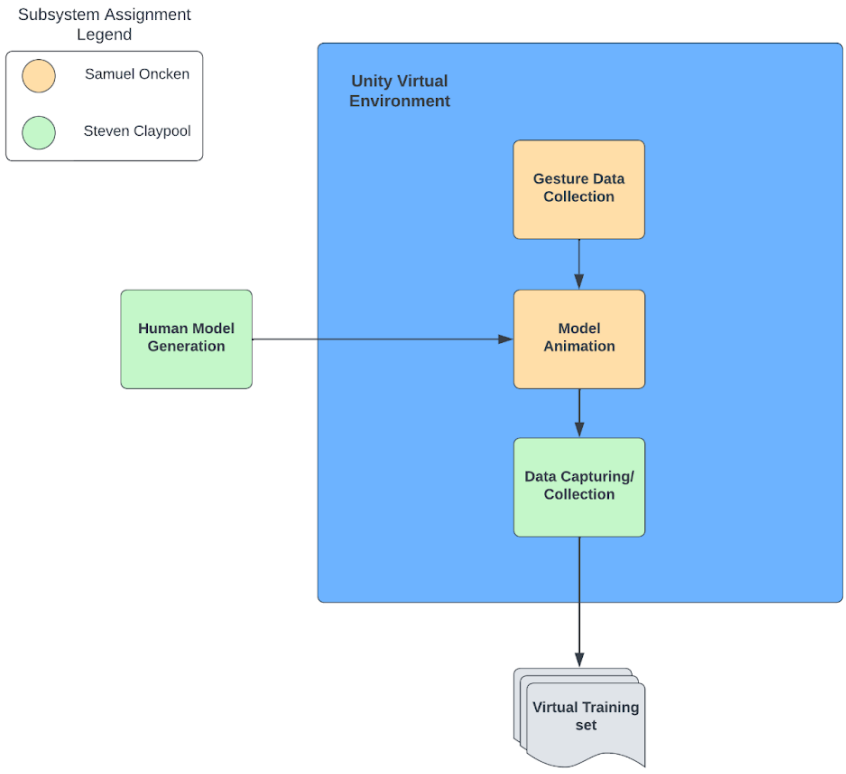


Figure 2. Block Diagram of System

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 image classification 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. For object detection, the accuracy metric of mean average precision (mAP) will be used.

> *Rationale: For a comparison to be valid, the performance metric of each test must be the same. The gesture recognition accuracy or mAP 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.

3.2.4.8 TensorFlow 2 Object Detection API
The latest object detection folders for TensorFlow 2 from TensorFlow's official models GitHub repository.

Rationale: The object detection that will be run for additional image analysis will use the TensorFlow 2 Object Detection API for running a Faster RCNN.

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 - 3
21 April 2023

INTERFACE CONTROL DOCUMENT
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72 4/21/2023

Author Date

APPROVED BY:

Samuel Oncken 4/21/2023

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
3	4/21/2023	Samuel Oncken Steven Claypool		Release of 404 Final Report

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.

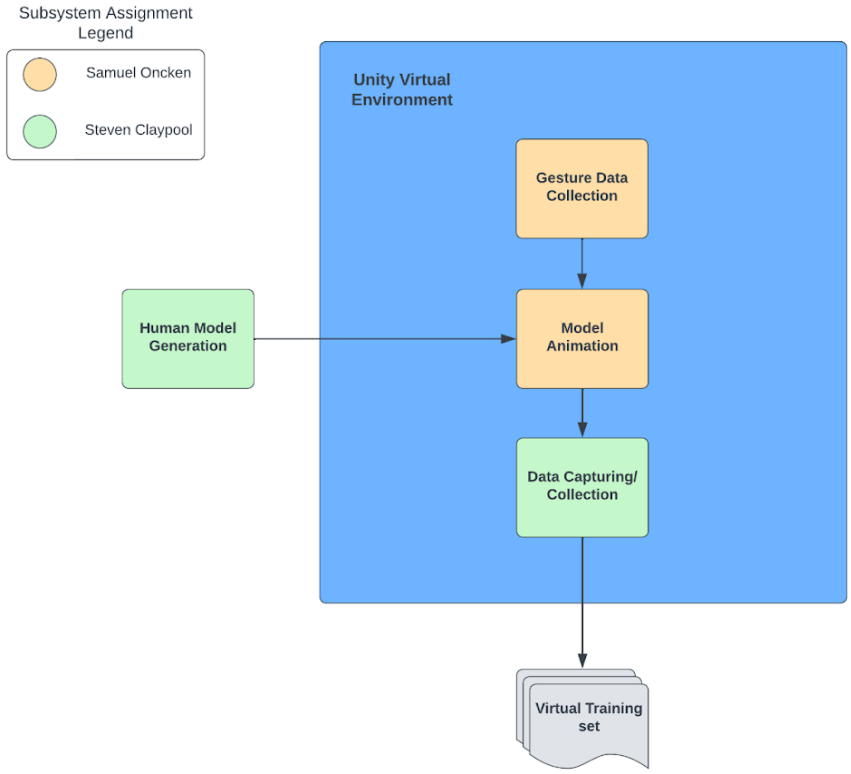


Figure 1: Block Diagram of subsystem interaction

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.

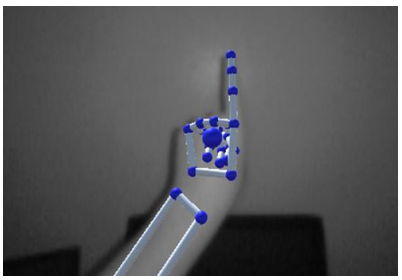


Figure 2: Hand Mapping Through LMC Control Panel

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.

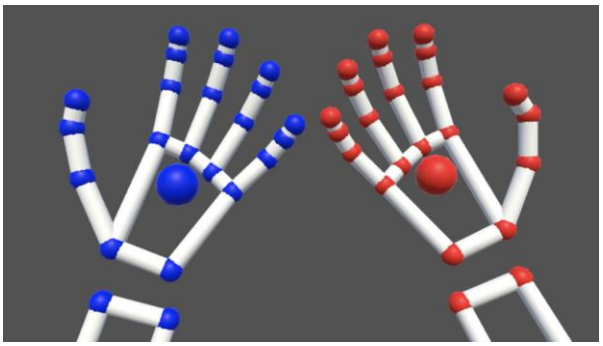


Figure 3: Ultraleap Rigged Hand Model in Unity

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.*

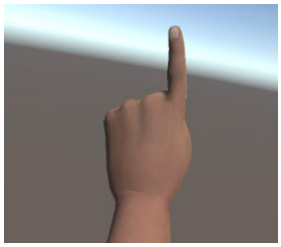


Figure 4: LMC Tracking directly to MakeHuman Model in Unity

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.

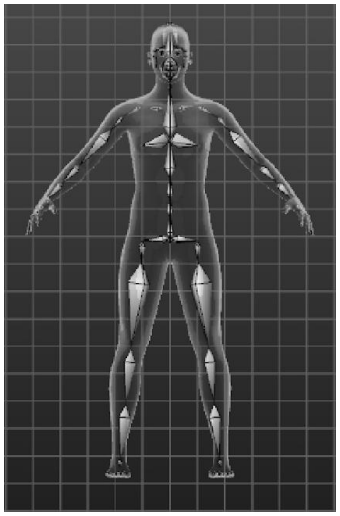


Figure 5: MakeHuman default rig for a human model

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.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

EXECUTION AND VALIDATION PLAN

REVISION - 3
15 April 2023

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
3	4/15/2023	Samuel Oncken Steven Claypool		Addition of 404 Execution/Validation Plan

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	Pranav Dhulipala
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

Table 2: 404 Validation Plan

Test Name	Success Criteria	Methodology	Status	Responsible Engineer(s)
Faster RCNN Validation	Create a functional Faster RCNN that takes images as an input and outputs the image with correct bounding boxes	Following working model examples, feed sample images into the Faster RCNN and verify correct output of images with bounding boxes	TESTED - Pass	Steven Claypool

Faster RCNN Object Detection on Real Data	Test the Faster RCNN with the HANDS dataset and reach acceptable accuracy metrics	Feed real dataset into Faster RCNN and measure accuracy using mAP (mean average precision)	TESTED - Pass	Steven Claypool
Faster RCNN Object Detection on Synthetic Data	Test the Faster RCNN with a synthetic HANDS dataset and reach similar accuracy compared to the real dataset	Feed synthetic dataset into Faster RCNN and measure accuracy using mAP (mean average precision)	TESTED - Pass	Steven Claypool
Import and Use Mixamo Animations	Animation clips from Mixamo FBX imported files can be applied to MakeHuman virtual models,	Download FBX (for Unity) files from mixamo.com for animations we wish to use. Import them into Unity project folder. Make each prefab “Humanoid” and extract and test animation clip on MakeHuman models	TESTED - Pass	Samuel Oncken
Randomly Select “Behavior” Animation	After model is spawned, depending on user input for desired “behavior”, a randomly selected animation clip will play.	Use Unity C# to script the categorization of “behavior” specific animation clips and random selection. Use Animator on human model with an animation controller selected that includes all desired animation clips.	TESTED - Pass	Samuel Oncken
Ability to Chain “Behavior” Animations	While environment is running, user is able to manually create test behavior chains or system is able to automatically randomly generate behavior chains	Use Unity C# to script the choice of a “Test Mode” vs. automatic mode. In Test mode, user can manually create behavior chains which link animations clips one after the other in order established by the user. Automatic mode uses a method in Unity C# to randomly generate chains of behavior animations.	TESTED - Pass	Samuel Oncken
Virtual Robot Unity Environment Integration	Be able to merge the testing environment with the robotic arm simulation environment created by Pranav. Make sure all functionality is still available.	Transfer all necessary code, model prefabs, unity scenes, etc.through GitHub Desktop and integrate all of these into the environment already created by Pranav with included robotic arm model.	TESTED - Pass	Samuel Oncken
Human Model Interaction with Robotic Arm Model	If criteria is met such as human models becoming too close to robot arm, they perform a randomized reaction animation	Unity C# scripting for reaction animations to take place using a trigger collider around the robotic arm prefab. Once a model comes into contact with the trigger, all other behaviors in the behavior chain cease to run and a reaction animation is immediately played.	TESTED - Pass	Samuel Oncken

Execution Plan

Table 3: Execution Plan Status Indicator Legend

Complete	
In progress	
Planned	
Behind Schedule	

Table 4: 403 Execution Plan

Task	Deadline	Responsibility	Status
Develop research outcomes/goals	9/7/2022	All	
Get familiar with LMC, Unity, and MakeHuman	9/14/2022	All	
Write ConOps	9/15/2022	All	
Determine optimal mounting position of LMC with testing of tracking accuracy	9/21/2022	Samuel Oncken	
Generate 4 test MakeHuman models, instantiate them into Unity	9/21/2022	Steven Claypool	
Map movement of hands onto Ultraleap hand prefabs in Unity	9/28/2022	Samuel Oncken	
Add rig settings and animator component to humans in Unity, test online example animation on models	9/28/2022	Samuel Oncken	
Set virtual cameras in Unity to face human model upon spawn	9/28/2022	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	4/24/2022	Steven Claypool -> Pranav Dhulipala	
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	

Table 5: 404 Execution Plan

Task	Deadline	Responsibility	Status
Set 404 Goals and Establish Project Plan	1/15/2023	All	
Import Mixamo Animations and Apply to MakeHuman Models	1/20/2023	Samuel Oncken	
Status Update Presentation 1	1/30/2023	All	
Build Customizable, Individual Behavior Modules	2/5/2023	Samuel Oncken	
Setup the RCNN and validate the function of the individual parts of the basic object detection, OpenCV selective search, and final RCNN detection.	2/12/2023	Steven Claypool	
Status Update Presentation 2	2/13/2023	All	
Build HANDS Dataset for RCNN Analysis	2/14/2023	Samuel Oncken	
Test and validate the training and evaluation of the real HANDS dataset on RCNN	2/19/2023	Steven Claypool	
Test and validate the training and evaluation of the synthetic HANDS dataset on RCNN	2/26/2023	Steven Claypool	
Gain Ability to Chain Behavior Modules. Transition Between Behavior States	2/26/2023	Samuel Oncken	
Automate Behavior Chain Generation and Performance	3/1/2023	Samuel Oncken	

Status Update Presentation 3	3/1/2023	All	
Integrate Fully Functional Training Environment with Pranav's Virtual Robot Environment	3/22/2023	Samuel Oncken	
Status Update Presentation 4	3/22/2023	All	
Continue analysis using different CNN model architectures within the RCNN. Troubleshoot any issues with the model.	4/3/2023	Steven Claypool	
Debug and Adjust Full System with Functional Robotic Arm	4/3/2023	Samuel Oncken	
Status Update Presentation 5	4/3/2023	All	
URS Thesis Report	4/3/2023	All	
Setup Faster RCNN and validate the training and testing of example data	4/10/2023	Steven Claypool	
Test and validate the training and evaluation of real HANDS dataset on Faster RCNN	4/14/2023	Steven Claypool	
Validate Full System Functionality and User-Input Checks Begin Work on Gripper Issues	4/14/2023	Samuel Oncken	
Final Presentation	4/17/2023	All	
Build HANDSv2 Dataset for Faster RCNN Analysis	4/18/2023	Samuel Oncken	
Test and validate the training and evaluation of synthetic HANDS dataset on Faster RCNN	4/19/2023	Steven Claypool	
Solve Gripper Issues – Develop Animation to Control Gripper Movement	4/25/2023	Samuel Oncken	
Run various training and evaluation combinations, testing on similar data as well as testing real training tested on synthetic and synthetic training tested on real	4/25/2023	Steven Claypool	
Final Demo	4/25/2023	All	
Complete Final Report	4/30/2023	All	

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

SUBSYSTEM REPORTS

REVISION - 2
15 April 2023

SUBSYSTEM REPORTS
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72 4/15/2023
Author Date

APPROVED BY:

Samuel Oncken 4/15/2023
Project Leader Date

John Lusher, P.E. Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	11/28/2022	Samuel Oncken Steven Claypool		Release for Final Report
2	4/15/2023	Samuel Oncken Steven Claypool		Release for 404 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 their desired dataset. 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 as described in the previous section. All instructions for use of this subsystem have been written in a GitHub repository which will act as a user instruction manual. The gesture data collection subsystem outputs unique animation clips for future use in the model animation subsystem. For this research, the gesture recordings are designed to match existing, real dataset gesture performances to compare dataset recognition accuracy results in future analysis. Our virtual environment will be used to replicate three existing datasets: Sign Language for Numbers, American Sign Language, and HANDS datasets.

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, they can be placed into Animation Controllers and therefore be scripted to play at random on imported virtual human models in future subsystems. This subsystem is the root of the dataset generation system as without it, uniquely identifiable gesture animations would not exist, and a hand gesture dataset could not be created. Due to this, this subsystem has been tested and altered 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 was determining the best method of recording gesture animations while planning for animation application in future subsystems. Initially, gestures were recorded (and exported as .fbx files) on a sample hand prefab provided by Ultraleap. These prefabs contained the necessary scripts already written and configured by the Ultraleap developer team to simplify use. However, using this method, the hand bone transformation data contained in the created animation clip (within the exported .fbx file) led to compatibility issues with MakeHuman models. 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 those within the MakeHuman models (shown in Figure 2) which were to be used in animation application. These differences resulted in the animation clips not being able to play on MakeHuman models. In addition, Figure 3 shows that when exporting the gesture animation as a .fbx file, the animation clips that came with the file

contained re-named bones, where each instance of '.' was automatically renamed as '_' (e.g., "wrist.L" turned into "wrist_L"), which once again resulted in naming incompatibility and animation application issues.

Figure 2: MakeHuman Default Rig Bone Labeling

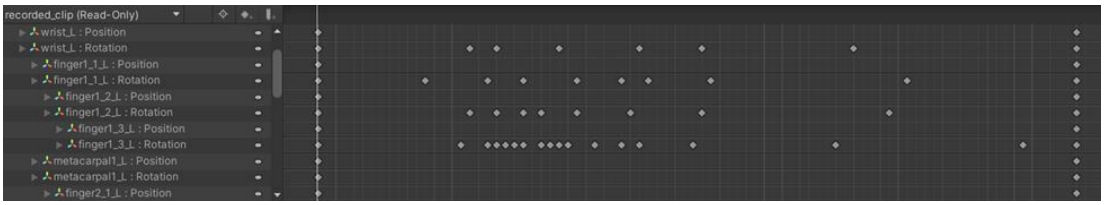


Figure 3: Animation clip using FBX Exporter - Not Compatible

The solution to this problem was to map Leap Motion hand tracking data directly to a MakeHuman model (shown in Figure 4) and export as an animation clip alone, which resulted in animation clip keyframes similar to that displayed in Figure 5 that can be directly applied to freshly imported MakeHuman models.

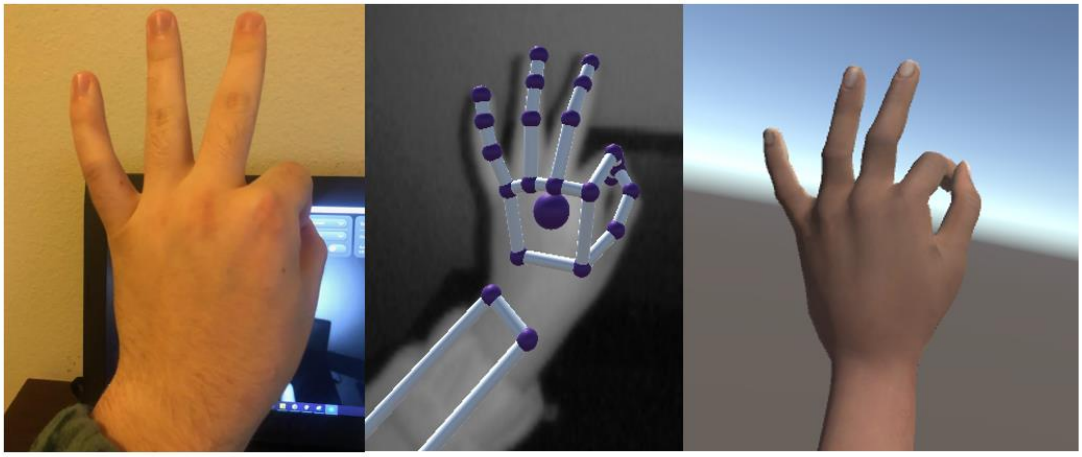


Figure 4: Recording of Animation Clips Directly on MakeHuman Model

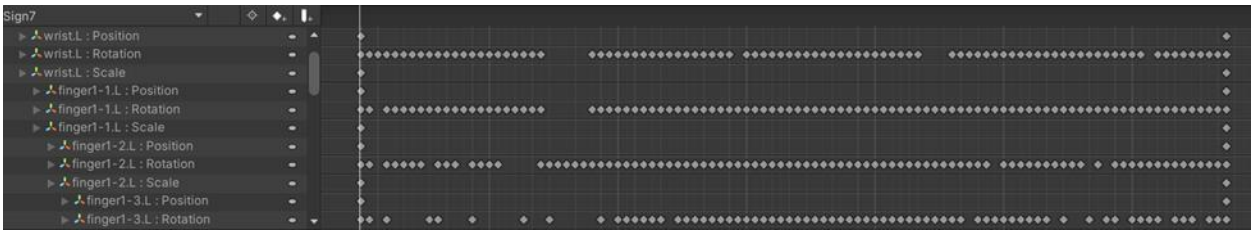


Figure 5: Animation Clip using Finalized Method - Compatible and Accurate

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 new MakeHuman model. To test the model animation application, a new Unity scene was created where a freshly generated MakeHuman model with a “Default” rig would be imported. An Animator component was then added to the “lowerarm02.L” bone of the model since each animation was recorded from the lowerarm02. A sample animation clip was then placed into an Animation Controller, shown in Figure 6, which controlled the Animator component on the model. Once the animation was able to be applied correctly and stop its motion in the final hand gesture position as desired, the subsystem proved to be applicable to future subsystems.

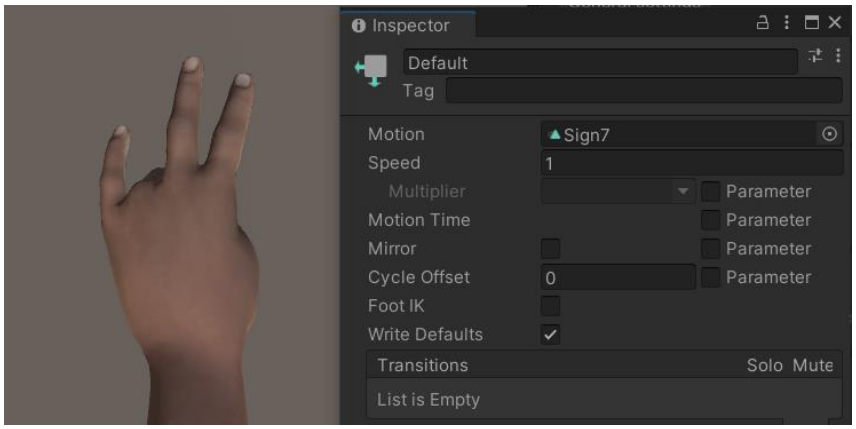


Figure 6: Test Application of Sign7 Gesture Animation Clip

In addition, the use of the Leap Motion Controller was validated within this subsystem by testing various LMC mounting positions under different lighting conditions to see which resulted in the highest tracking accuracy upon inspection. It was confirmed that by head mounting the LMC, the best motion tracking was achieved for the gestures that were planned to be recorded. Many of the gestures within the real hand gesture datasets involved putting one or more fingers down (into palm region) while still holding up others. It had to be ensured that the LMC could recreate this type of movement accurately. Figure 7 depicts the inaccuracy of recording this type of motion using Desktop Mounted mode as opposed to Head Mounted mode.

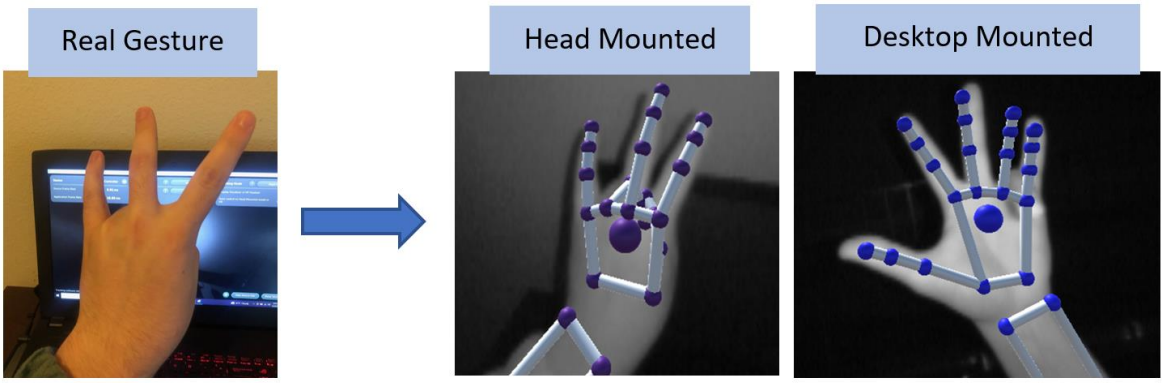


Figure 7: Tracking Accuracy of Head Mounted LMC vs. Desktop Mounted LMC

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 created Unity script. 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, shown in Figure 8, and MakeHuman's code and running into multiple issues with the process, it was determined that it would require extensive work to make such changes and as a result, MakeHuman models were generated using 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, shown in Figure 9. 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.

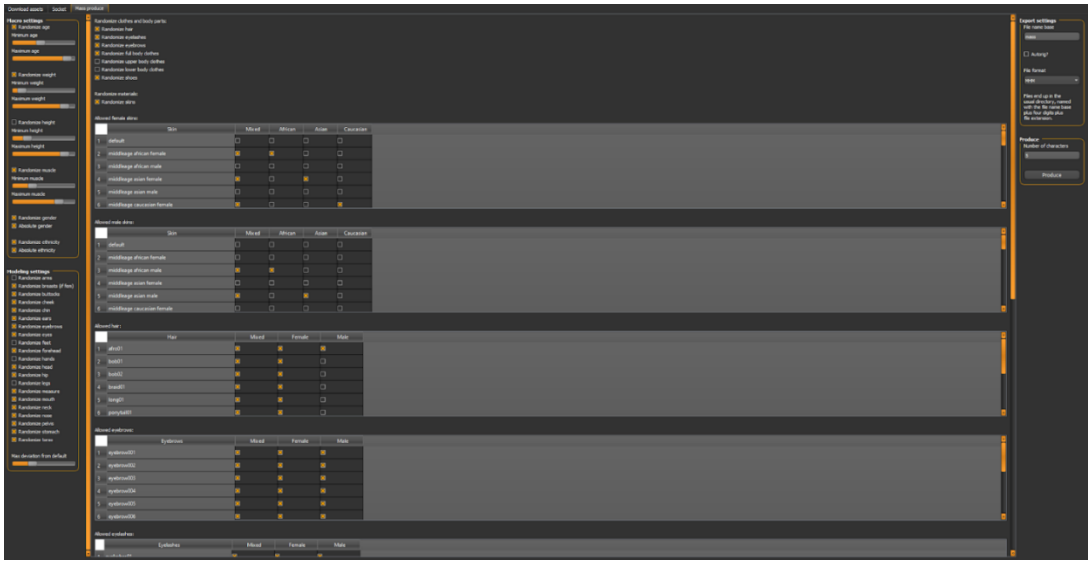
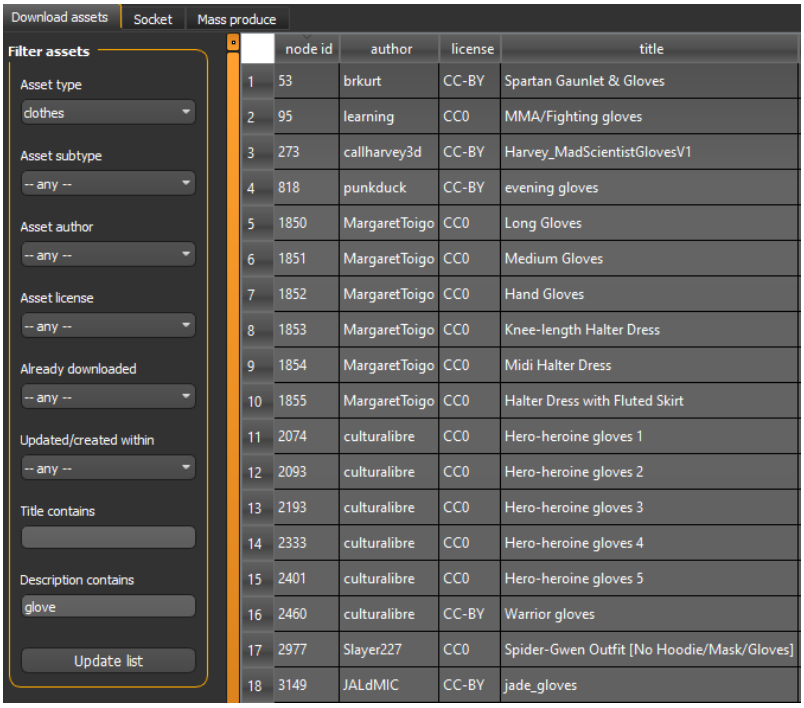


Figure 8: Mass Produce Plugin Page



	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

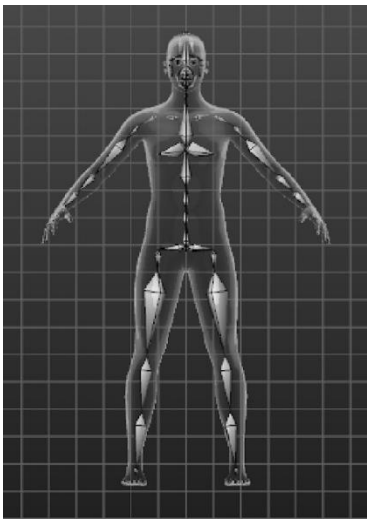


Figure 10: Default Rig for MakeHuman Models

For the loading and random generation of the human models, functions within the scene controller script were written and are shown in Figure 11. The model loading method was provided by Pranav Dhulipala; 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 a gesture animation to be performed. 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, HANDS, 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, adding an Animator component to the lowerarm02 bone and an Animation Controller for the Animator to use would be done manually. A singular animation clip would be placed manually into the Animation Controller to test performance as shown in Figure 12.

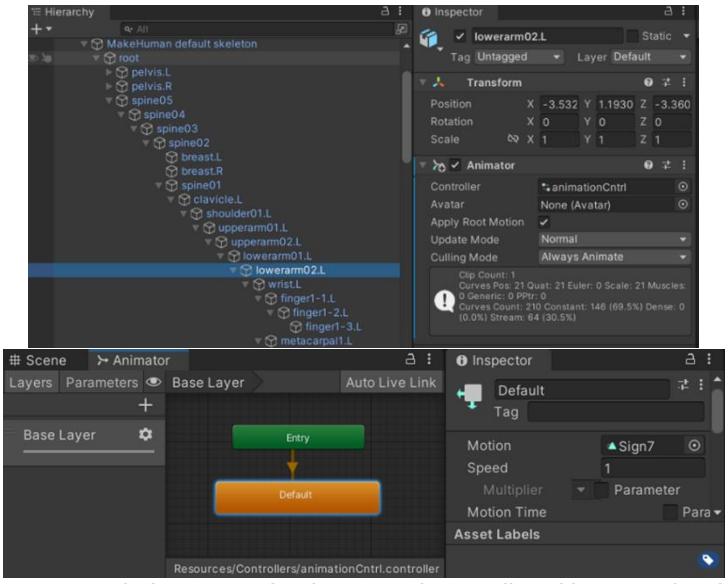


Figure 12: Manual Placement of Animator and Controller with One Animation Clip

However, this approach had to be automated after the Human Model Generation subsystem was implemented since freshly spawned models are being instantiated consecutively into the scene without these components on a constant time interval. To begin implementation of this change, 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, as shown in Figure 13. 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.

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

The next automation steps must be implemented through script. Once a MakeHuman model has been instantiated into the scene, the first scripted step is to locate the lowerarm02.L bone as shown in the figure below.

```
//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 to the root bone of motion (lowerarm02.L) and enable the component. 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. A portion of this scripted process is displayed in Figure 15.

```
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

Optimization for the time it would take to build the entire datasets was implemented 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 had been performed the maximum number of times selected, the data capturing/collection subsystem would stop the running of the Unity environment. Figure 16 displays the coded implementation of this optimization.

```
//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. The automation of adding components/controllers to sequentially spawned models was then implemented, which was tested and validated on its own using an independent "ModelAnimation" script (which was then incorporated into the finalized SceneController script in the completed Unity environment). Figure 17 depicts the performance of the gesture animation K on 3 unique, consecutively 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

It was 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. It was 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, it was validated that the subsystem would remove gestures from the pool of possible animations 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, the new string of choices was printed, and it was observed that the choice of the particular gesture had been removed as seen in Figure 18 and Figure 19. Table 1 summarizes all tests used to validate the model animation subsystem.

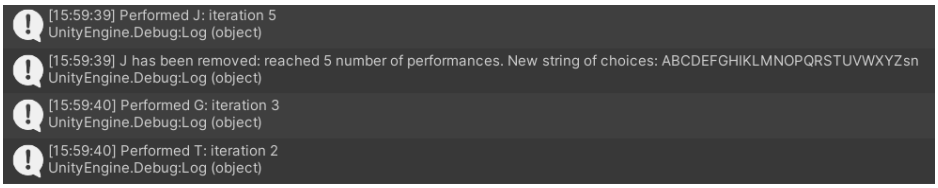


Figure 18: Log After First Letter J Reached Maximum Number of Iterations

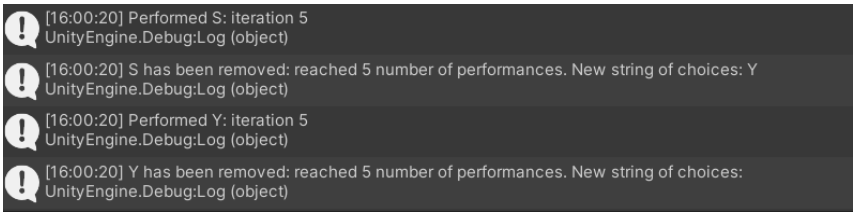


Figure 19: Log After Last Letter Y Reached Maximum Number of Iterations

Table 1: Overview of all Tests Performed to Validate the Model Animation Subsystem

Test Name	Test Summary	Test Result
Hand Gesture Application	When the Unity environment is run, a singular recorded hand gesture animation can be performed on a MakeHuman model.	Pass
Auto Animator Placement	When the Unity environment is run, each spawned MakeHuman model (with no Animator component on its lowerarm02 bone prior to run) contains an automatically placed (through script) Animator component on the lowerarm02 bone and can perform animations.	Pass
Recorded Gesture Animation Functionality	Each recorded hand gesture animation clip may be applied and performed accurately on a MakeHuman model with a “Default” rig.	Pass
Gesture Performance Amount	Each hand gesture animation clip is performed only until a user-specified number is reached.	Pass
Gesture Performance Optimization	After a hand gesture animation clip is performed the user-specified number of times, it is removed from the potential choices of animation clips to be performed.	Pass

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. It then stores the image as a .jpg in folders sorted by dataset and gesture using a scripted “TakeImage” coroutine. 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 more diverse data. Images are sized to be 512x512 pixels for neural network usage.

5.2. Subsystem Details

This subsystem works in tandem with the model animation subsystem, as both the PlaceCamera and TakeImage functions are called directly after a random animation is performed. For this subsystem to function properly, many virtual cameras had to be placed within the scene hierarchy that faced each 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 Figure 20. This is because all gestures were recorded using the exact same head mounted LMC positioning and virtual arm placement in the gesture recording stage. Since the gesture G required the hand to be pointing towards the right of the scene view, it was necessary to alter the camera angle when looking at the played animation to make the gesture look as though it was recorded with a horizontally orientation.

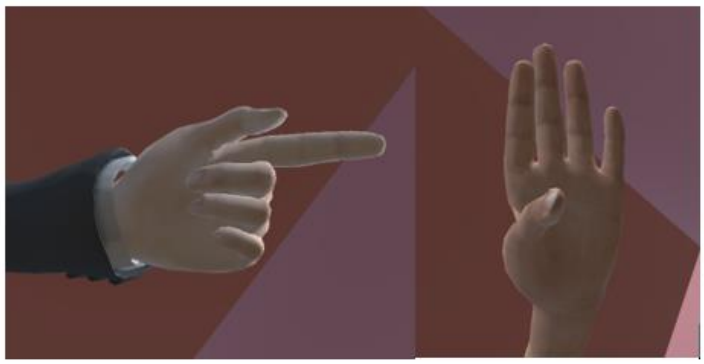


Figure 20: Camera Placement for Gesture G vs. B

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, as shown in Figure 21. By adding randomness in the camera position, it is ensured that the gesture will be viewed from several 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 gesture performance in game view, naming the output file, and designating the output folder path depending on the chosen dataset and gesture animation. The Sign Language for Digits dataset named its files using the structure: “name of the gesture_image number of that gesture”. 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 identifies the correct naming convention, names each image, and determines where to store the image depending on the signed gesture. A portion of the TakeImage coroutine is displayed in Figure 22.

```
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 a screenshot is taken using the enabled camera from the PlaceCamera script, the TakeImage coroutine introduces a slight delay. This delay is necessary because all animation clips begin in the same position, but over time move differently to form the final gesture sign. If a screenshot were recorded at the instant that a gesture animation clip was played, 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, each animation clip was sped up by a factor of at 1000 and the TakeImage coroutine implemented a delay of .02 seconds so that it would be ensured 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 number 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. Figure 23 shows how this is implemented.

```
//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 several tests. First, it was ensured that the images were taken at the correct time after a gesture animation was performed, resulting in photos of the gestures looking as intended (as explained in the previous section). It was determined that by delaying .02 units of time after the gesture was performed (while gestures were sped up 1000x), all screenshots were successful in recording data only after the full animation clip played out and the hand was in its final position, as seen in Figure 24. This resulted in accurate gesture images and a decreased amount of time required to play/record all gesture animations the maximum number of times.

Figure 24: Example Validation for Images of Completed Gestures

It was also validated that each gesture image was stored in the correct folder according to the dataset it belonged and named following the same convention as the real dataset used. To test each of these, the system was run using a maximum number of 5 images per gesture for all desired datasets. The system was allowed to play until it completed all required gestures and stopped itself. Referring to the folders that contained the output files, it was observed that each labeled folder contained images of the gesture that it specified (e.g., folder labeled one stored all images of one being portrayed). In addition, it was validated that each image was named correctly (e.g., images of one being portrayed were named “one…” for Sign Language for Numbers) and included a counter indicating the image number (up to 5 in this case), as demonstrated in Figure 25.

Figure 25: Example Validation of Images Properly Named and Stored

Finally, it was validated that the subsystem would stop the running of the Unity environment once all desired gestures were performed. This was tested by running the scene multiple times, each with a different number of desired images per gesture. During each run, an output message was displayed in the console once all gestures had been performed the maximum number of times as specified by the user. After this condition is satisfied, it was verified that the Unity environment would exit runtime. Referring to the dataset storage location, it was confirmed that all gesture folders contained the desired amount of gesture images, proving that the Unity environment stopped its run only after the desired dataset was fully generated. Table 2 summarizes all tests performed to validate the Data Capturing/Collection subsystem.

Table 2: Overview of all Tests Performed to Validate the Data Capturing/Collection Subsystem

Test Name	Test Summary	Test Result
Gesture Image Accuracy	After the Unity environment is run, gestures in each output image are performed accurately and are as desired by the dataset creator.	Pass
Image Storage Accuracy	After the Unity environment is run, each image is stored in the dataset folder that it belongs and is inside of its correctly labeled gesture folder within the dataset.	Pass
Image Naming Accuracy	After running the Unity environment, all gesture images are properly named according to the real gesture dataset that was replicated.	Pass
Exit Functionality	After the Unity environment has performed every gesture within a dataset to the user-specified maximum number, the Unity player stops. Each gesture folder within the replicated dataset contains only the user-specified maximum number of images.	Pass

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.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

FULL SYSTEM REPORT

REVISION - 1
15 April 2023

FULL SYSTEM REPORT
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72 4/15/2023
Author Date

APPROVED BY:

Samuel Oncken 4/15/2023
Project Leader Date

John Lusher, P.E. Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	4/15/2023	Samuel Oncken Steven Claypool		Release for 404 Final Report

Table of Contents

List of Tables

List of Figures

1. Dataset Generation Full System Description

1.1. 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 with completely virtual data. 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), a Sign Language for Numbers dataset (11 gestures), and a HANDS dataset (12 gestures) consisting of 12,000 images per gesture. We have also more recently completed the generation of a HANDSv2 dataset (11 gestures) that contains full body images of HANDS gesture performances.

1.2. System Details

As described in each of the subsystem reports in the Methods section of this paper, each subsystem proved to be compatible with one another. 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 Unity exit functionality 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 their desired dataset size and can begin dataset generation by playing the Unity scene.

To provide a brief overview of the dataset generation system process after gesture animation clips have been recorded, 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 a random animation from the chosen dataset 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 name is determined, the folder path is selected, and a screenshot is captured and stored. This process repeats until all animations have been played a user-selected maximum number of times and the Unity environment stops running. At this point, the finalized dataset is created.

1.2.1. Additional System Characteristics

1.2.1.1. Randomization in Background Conditions

Background image randomization has been incorporated into our environment to increase diversity in our virtual datasets. To incorporate the written background randomization script into our virtual dataset generation environment, wall objects were created around the location where the models would be spawned. By placing the wall objects into the public variable “Walls” array within a created background randomization script, upon running the environment, each wall will appear as a random image from the Describable Textures Dataset, as shown in Figure 1.

Figure1: Background Image Randomization within Images of Gesture A

1.2.1.2. 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. The goal is to replicate real-world scenarios as best as possible, where lighting conditions are not always optimal, and hands might appear darker or brighter than desired as displayed by Figure 2. We plan for a gesture recognition neural network trained using our synthetic data to recognize images in various conditions.

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

1.3. System Validation

The final system was validated first by running the Unity environment under various user-selected conditions. The Unity dataset generation system was tested using the Sign Language for Numbers dataset with a selection of 5, 10, 50, 1,500, and 12,000 images per gesture. After each run, it was validated that each gesture image was accurately recorded, stored in the correct labeled gesture folder within the dataset folder, and was named according to the convention of the real gesture dataset that was replicated. It was also validated that each gesture image clearly contained the hand gesture without too much hand/skin cut out of frame. After the Sign Language for Numbers dataset passed all validation tests, the same trials were

run while replicating the American Sign Language alphabet and the HANDS datasets. For both additional dataset replication cases, all tests passed as all images were accurately taken of each gesture and stored in the correct folder paths, as displayed in Figure 3 with the gesture for the letter T. The 12,000 image per gesture dataset test is used as the final form of the replicated synthetic dataset set that is experimented with in training and testing.

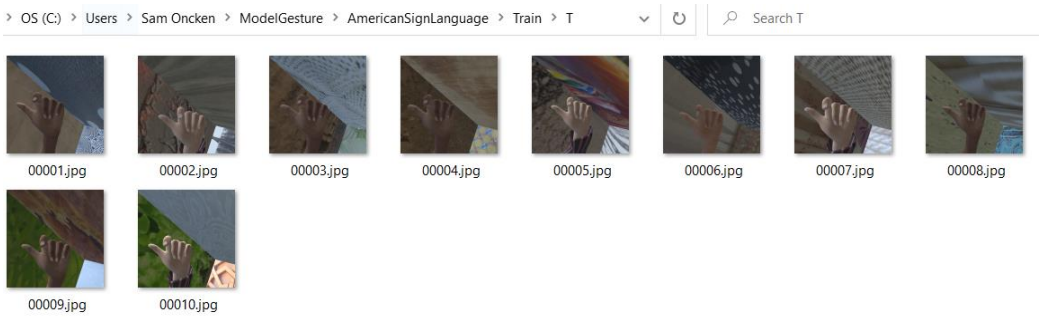


Figure 3: Validation Example - Letter T Performed and Stored 10 Times

The scene customizations were also validated by running a “RandomTexture.cs” script (random background image script) alone to see whether each wall changed appearance. While running both scripts (RandomTexture and SceneController) at the same time, it was determined that the best rate at which the scene background should change was .12 units of time, which allowed enough time for a new background to appear after each iteration of a model being spawned.

Finally, the light source randomization was validated 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. Table 1 summarizes all tests performed to validate the Dataset Generation System.

Table 1: Overview of Tests Performed to Validate the Dataset Generation System

Test Name	Test Summary	Test Result
Dataset Generation	After each run of the Unity environment, all gestures within the selected dataset being replicated are performed the user-selected maximum number of times and all images are stored correctly with correct naming conventions.	Pass
Background Randomness	During the running of the Unity environment, each plane surrounding the MakeHuman model changes appearance upon a new character being spawned.	Pass
Light Intensity Randomness	During the running of the Unity environment, light source intensities change every time a new MakeHuman model is spawned into the scene. Output images contain noticeably different lighting/shadow characteristics.	Pass

1.4. Dataset Generation Output

1.4.1. American Sign Language Dataset Replication

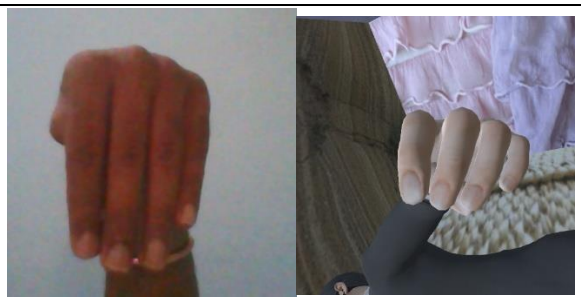
The American Sign Language Dataset is comprised of 28 distinct, static gestures: 26 letters of the English alphabet, a gesture for “Space”, and a gesture for “Nothing”. A fully synthetic version of the American Sign Language Dataset has been generated. Table 2 depicts each real dataset hand gesture in comparison with the synthetically generated version.

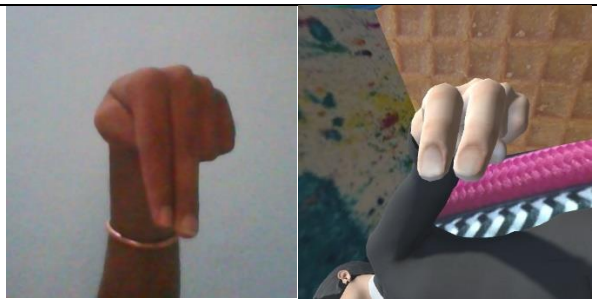
Table 2: Comparison of Real and Synthetic Gestures for American Sign Language Dataset.

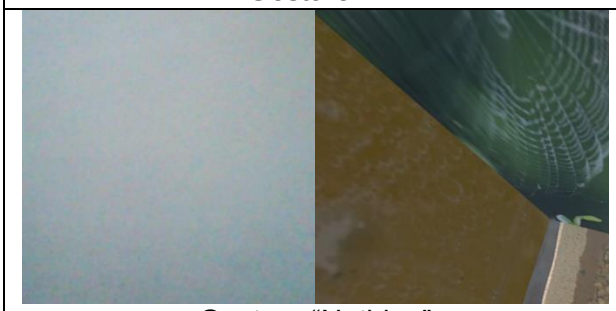
 Gesture A	 Gesture B
 Gesture C	 Gesture D
 Gesture E	 Gesture F
 Gesture G	 Gesture H
 Gesture I	 Gesture J


Gesture K


Gesture L


Gesture M


Gesture N


Gesture “Nothing”


Gesture O


Gesture P


Gesture Q


Gesture R


Gesture S

 Gesture “Space”	 Gesture T
 Gesture U	 Gesture V
 Gesture W	 Gesture X
 Gesture Y	 Gesture Z

Using the virtual environment developed in this research, a fully synthetic version of the American Sign Language dataset has been created. This virtual dataset is comprised of 336,000 images (12,000 images per gesture). Upon inspection of the example images in the table above, the gestures for the letter “D” are flipped in hand performance from real to synthetic data. All gestures implemented in the virtual environment were recorded using a left hand and therefore are stored as left-handed gestures. To account for this mismatch in this dataset or any other dataset being replicated, gesture images are mirrored at random during image preprocessing in the training of gesture recognition neural networks. By flipping images at random, the final gesture recognition model should recognize both right and left-handed gestures.

Certain hand movements were difficult to track/depict using the Leap Motion Controller, which led to gestures not appearing exactly as desired. . For example, looking at the gesture “W”, the pinky and thumb do not touch as they do in the real dataset image. This is a result of LMC’s tracking inaccuracy. A recurring issue faced in the recording of many gestures was the inability for the thumb to bend towards the palm of the hand. Fixing this issue was out of the scope of this project as it is a Leap Motion tracking/device issue. As a result, the recording of gestures was performed to the best accuracy as possible, and the screenshotted images have been stored and analyzed in a later section.

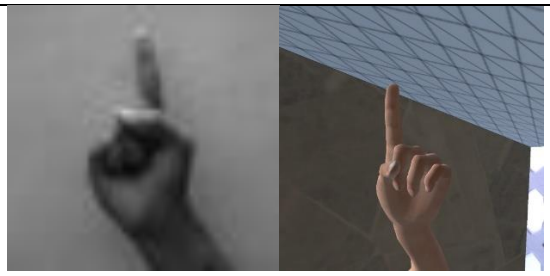
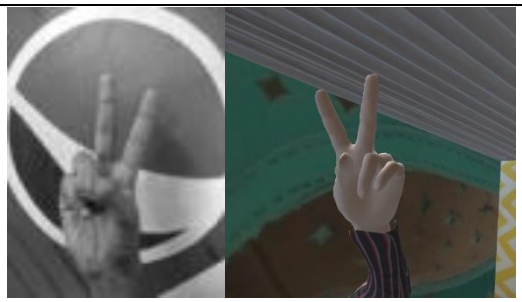
The largest inaccuracy in gesture recording can be seen in letters “M” and “N”. Both gestures required that the fingers bend and completely cover the palm region of the hand. This motion was not possible to obtain using the LMC, which resulted in poor gesture performance quality. To replicate the real gesture performance as closely as possible, virtual cameras were situated in specific locations so that the perspective makes it seem like the fingers are covering the palm.

There are a few more gestures in this dataset that were challenging to record due to LMC tracking errors, many of which are due to the problems listed above. A few examples include gestures “A”, “E”, and “S”, which all look very similar virtually due to these inaccuracies.

1.4.2. Sign Language for Numbers Dataset Replication

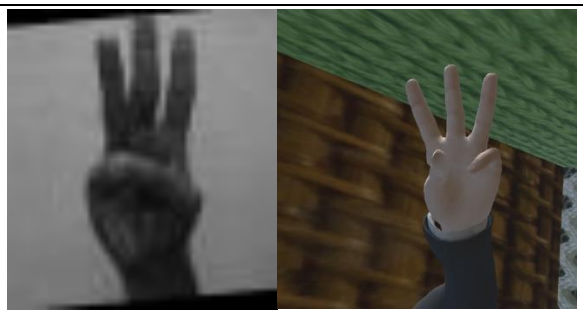
The Sign Language for Numbers Dataset is comprised of 11 distinct, static gestures: numbers 0-9 and a gesture for “unknown”. A fully synthetic Sign Language for Numbers Dataset has been generated using the Unity hand gesture dataset generation environment. Table 3 depicts each gesture from the real dataset in comparison to the synthetically generated gestures.

Table 3: Comparison of Real and Synthetic Gestures for Sign Language for Numbers Dataset.

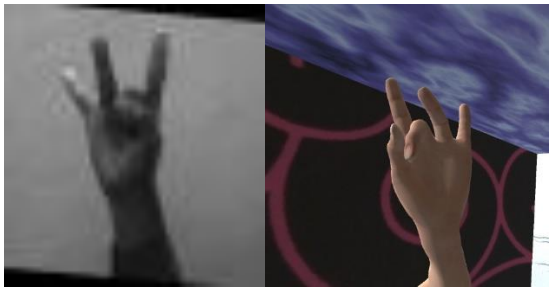
 Gesture 0	 Gesture 1
 Gesture 2	 Gesture 3

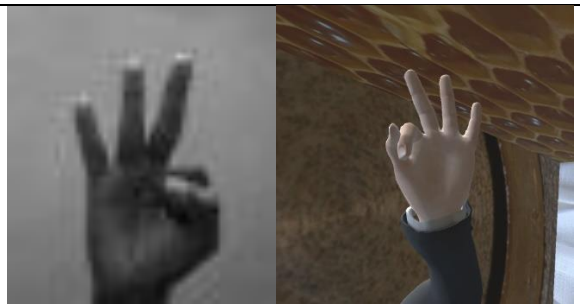

Gesture 4


Gesture 5


Gesture 6


Gesture 7


Gesture 8


Gesture 9

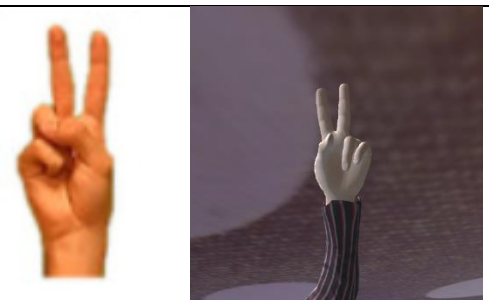

Gesture Unknown

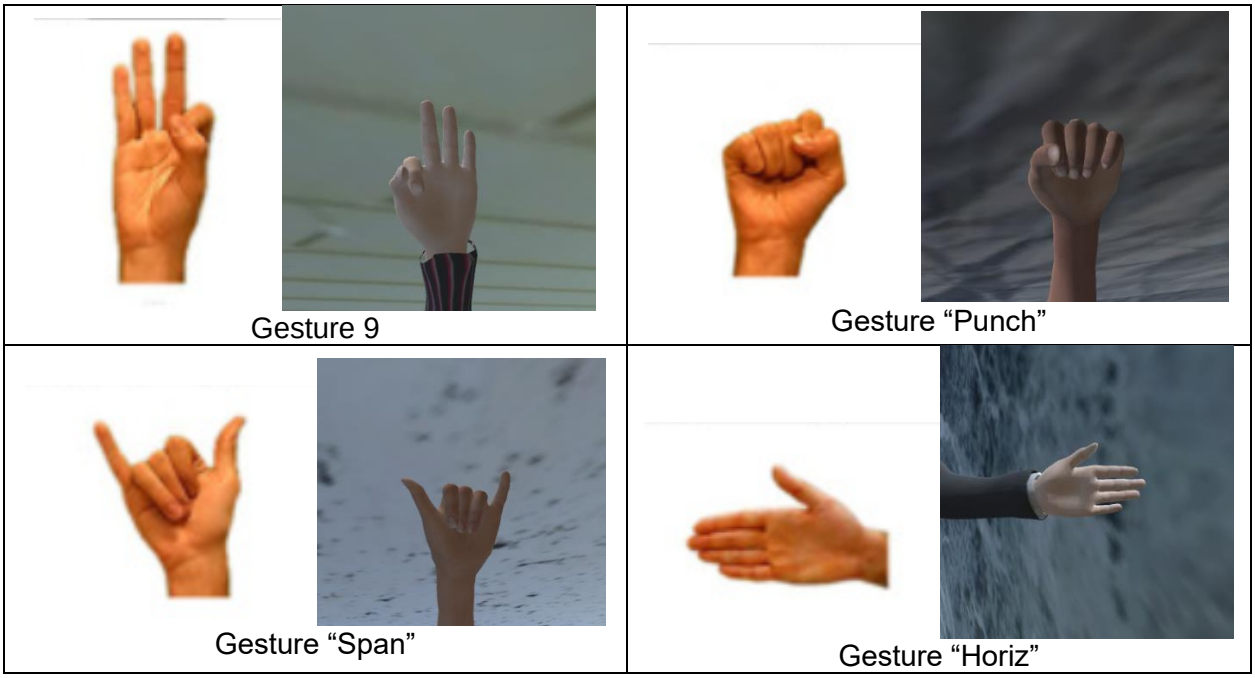
Using the virtual dataset generation environment, a synthetic Sign Language for Numbers dataset has been created, consisting of 132,000 images (12,000 per gesture). Once again, the fact that the virtual images are left-handed gestures while the real images are right-handed gestures is irrelevant as a random number of synthetic images will be flipped during the image preprocessing in the training of a gesture recognition neural network for the model to recognize both left and right-handed gestures. In addition, the LMC tracking inaccuracies discussed in the American Sign Language Dataset Replication section caused problems for a few gestures such as the numbers “6” and “8”. Nevertheless, the recognition accuracies of gesture recognition models trained on this dataset will be evaluated in a future section of this paper.

1.4.3. HANDS Dataset Replication

The HANDS Dataset is comprised of 15 distinct, static gestures. However, only 12 of these gestures can be performed with one hand. These 12 gestures have been replicated synthetically and include: numbers 0-9, a gesture for “Punch”, a gesture for “Horiz”, and a gesture for “Span”. Table 4 compares each real, one-handed gesture from the HANDS Dataset to its synthetically generated counterpart.

Table 4: Comparison of Real and Synthetic Gestures for HANDS Dataset.

 Gesture 1	 Gesture 2
 Gesture 3	 Gesture 4
 Gesture 5	 Gesture 6
 Gesture 7	 Gesture 8



Using the virtual dataset generation environment, a synthetic HANDS dataset has been created, consisting of 144,000 images (12,000 per gesture). New animation recordings were used to depict each gesture within this dataset even though many of the gestures included in the HANDS dataset were similar to those in other completed datasets. This extra work was done to produce more high-quality gesture images. Another reason was to match the gesture performances to those within the real HANDS dataset more accurately as some gestures were performed differently than those within the Sign Language for Numbers dataset. Once again, image flipping will take place in the image preprocessing of training a hand gesture recognition neural network so that the model can recognize both right and left-handed gesture performances.

1.4.4. HANDS Version 2 Dataset Replication

After completely developing the virtual HANDS dataset depicted in the section above, it was planned for this dataset to be fed directly into the convolutional neural network layer within the RCNN for training. In this way, the model would be trained to recognize hands first, then be given proposed regions using OpenCV’s selective search during a full run of the RCNN. However, in the process of setting up the RCNN, it was concluded that a simple RCNN would not function as intended within this research. Instead, a faster RCNN was implemented. In the faster RCNN training process, the CNN cannot be trained separately from the faster RCNN using hand images. Instead, training is done with the faster RCNN classification and region proposal as a whole using images with ground truth bounding box labels around the hand. As a result, a new dataset needed to be generated which consisted of full body images with gesture performances. An example image of the version 2 dataset is depicted in the figure below.

Figure 4: Example HANDS v2 Output Images

Using the virtual dataset generation environment, a fully synthetic HANDS version 2 dataset has been generated and is currently being used in Steven's faster RCNN training. Slight modifications to the environment needed to be made to create this dataset. The first modification was to alter the bodily positions of the MakeHuman models to lower their right arms to be by their side and to raise their left arms in position to sign a hand gesture. Once these bodily alterations were made, the changed MakeHuman models were saved as prefabs in a separate folder from the original models. They were saved as prefabs so that the model would remember the arm/hand placements in space and be able to spawn directly into the scene already in position. The next step was to alter the HANDS camera placement to face the human model directly, as was done in the real dataset being replicated. The figure below compares the camera perspective from the real dataset and that of the virtual dataset.

Figure 5: Camera Angle Comparison of Real Dataset (left) and Virtual Dataset (right)

Five unique MakeHuman models were altered with correct hand placement to replace the five human performers used in the real dataset generation process. The last environmental modification involved removing the camera randomization so that each model gesture performance would be depicted in the same way in each output image. This was done so that Steven could more easily mass-produce bounding boxes around each model hand instead of manually drawing a bounding box around each randomized hand position, which would take far too long. The positional randomization will be re-implemented in the preprocessing of the faster RCNN training process, where random positional and angular changes can be made before being fed into the model. After each of these modifications was completed, the dataset was run using 2,000 images/gesture/model for each of the 5 models, resulting in 10,000 images/gesture and a grand total of 110,000 images in the dataset. The "Horiz" gesture was disregarded in this new full body process because in the original virtual HANDS dataset, the Horiz gesture was performed with the hand just facing upwards, but with the camera perspective making it seem

like the hand was angled to the left. This same methodology could not be implemented in a full body setting and the only viable method to implement this gesture would have been to create an entire new set of models with specific arm placements for the Horiz animation, which would have taken too long as we are approaching the end of the semester.

1.5. System Conclusion

The full system passed all validation tests, proving that all previously designed subsystems were integrated successfully, and the final dataset generation system had been completed. The finalized system was used to generate fully virtual versions of the Sign Language for Numbers dataset, the American Sign Language dataset, the HANDS dataset, and the HANDSv2 dataset containing 132,000, 336,000, 144,000, and 110,000 images respectively.

2. Dataset Training and Testing Validation

2.1. CNN Gesture Classification

2.1.1 Dataset Training Validation

Using the generated synthetic datasets, many different CNN models were trained to validate the use of the synthetic datasets in training image classification models. This process was also done to determine the most optimal training method, considering transfer learning, training from scratch, and using or excluding image augmentation. The training and testing results were obtained by our graduate student, Pranav Vaidik Dhulipala, using our system and generated datasets.

For the synthetic data training and validation in Figure 6 (left), a gradual learning curve is seen, indicating a well-trained model that reaches acceptable accuracy metrics. For the real data training and validation in Figure 6 (right), the significantly smaller dataset size led the model to overfit on the data very quickly. This shows the benefit of the virtual environment as the number of images per gesture can easily be increased to expand the dataset significantly.

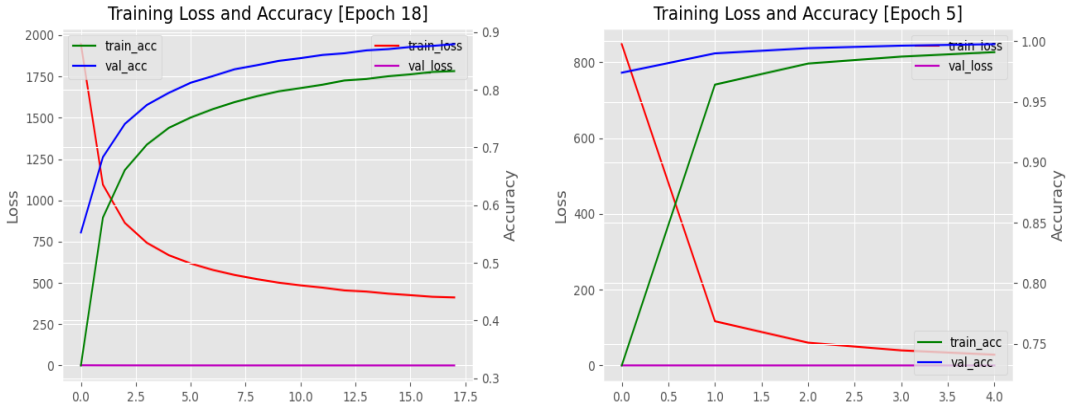


Figure 6: Inception CNN Architecture Trained with Transfer Learning and Image Augmentation for the Synthetic ASL Alphabet Dataset (left) and real ASL Alphabet Dataset (right)

For models trained on images without image augmentation, the general outcome was immediate overfitting of the model due to the lack of complexity in the image, as seen in Figure 7 (left). This lack of complexity also allowed the training and validation accuracy to almost match since the training data less difficult to learn. However, some cases showed different behaviors, like with the ResNet50 CNN architecture in Figure 7 (right) where the model still maintained a gradual learning curve but failed to reach a high enough accuracy. This change in behavior reinforces the idea of requiring extensive analysis of various combinations of methods and data to exhaust all possibilities.

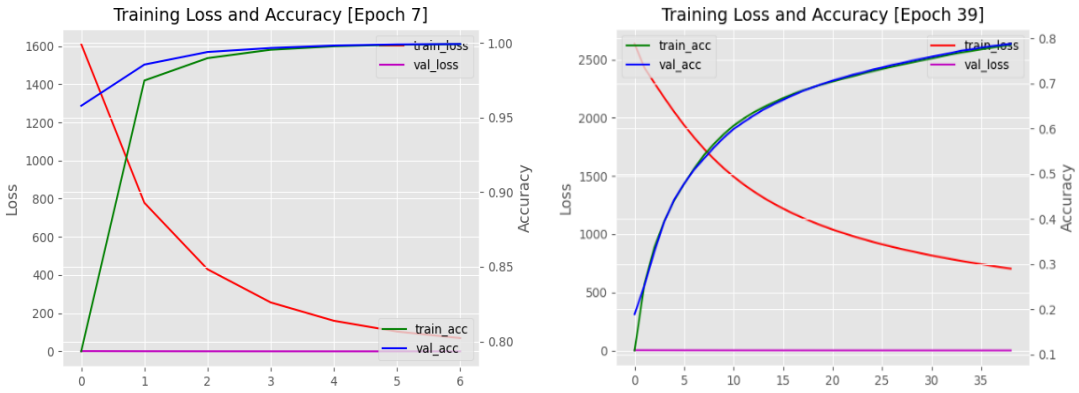


Figure 7: VGG19 CNN Architecture Trained on Synthetic ASL Alphabet Dataset (left) and ResNet50 CNN Architecture on Real ASL Alphabet Dataset (right) with Transfer Learning and Without Image Augmentation

As seen in Figure 8 (left), training with transfer learning again maintained a gradual learning curve that reached acceptable accuracy metrics. When training from scratch as seen in Figure 8 (right), the limitation of using a small dataset is clearly visible as the model quickly overfit. Without the pretrained parameters from ImageNet, which is a multi-million image dataset with 1000 different classes, the model overfits and would struggle to classify any images that differ from the training data.

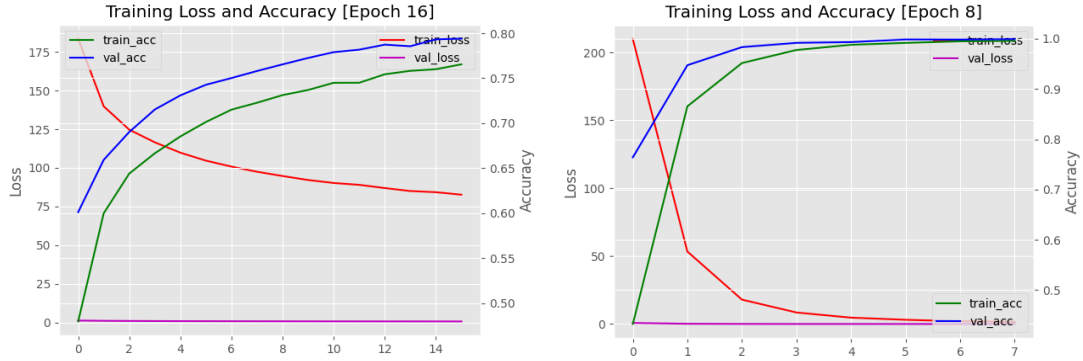


Figure 8: Xception CNN Architecture Trained on Synthetic ASL Numbers Dataset with Transfer Learning and Image Augmentation (left) and Real ASL Numbers Dataset from Scratch with Image Augmentation

From the analysis using various training methods, using transfer learning and image augmentation is determined to be the optimal training method for most model architectures. This was because the models trained using transfer learning and image augmentation generally avoided overfitting and gradually trained to a sufficient image classification accuracy, with most models ending with around 80% or higher validation accuracy. This final training accuracy of 80% or higher on models trained with synthetic data also validate the use of synthetic image datasets for training a CNN.

2.1.2 Optimal CNN Model Architecture Validation

Then, to verify which CNN model architectures performed best with the image data used, further analysis was performed for the following model architectures: Inception, ResNet50, VGG16, VGG19, and Xception. Comparisons between the training for the real and synthetic alphabet dataset are shown in Figures 9, 10, 11, 12 and 13 below. For training with the real alphabet dataset, most of the models trained to 100% accuracy rapidly, which is a sign of overfitting, with just ResNet50 barely reaching 70% validation accuracy after 39 epochs. However, the overfitting is expected, since the real ASL Alphabet dataset is much smaller than the synthetic dataset and lacks complexity. For training with the synthetic ASL Alphabet dataset, Inception, ResNet50, VGG19, and Xception trained to near or above 80% validation accuracy, while VGG16 only reached 50% validation accuracy. The models trained gradually, which is likely due to the larger dataset size and increased image complexity.

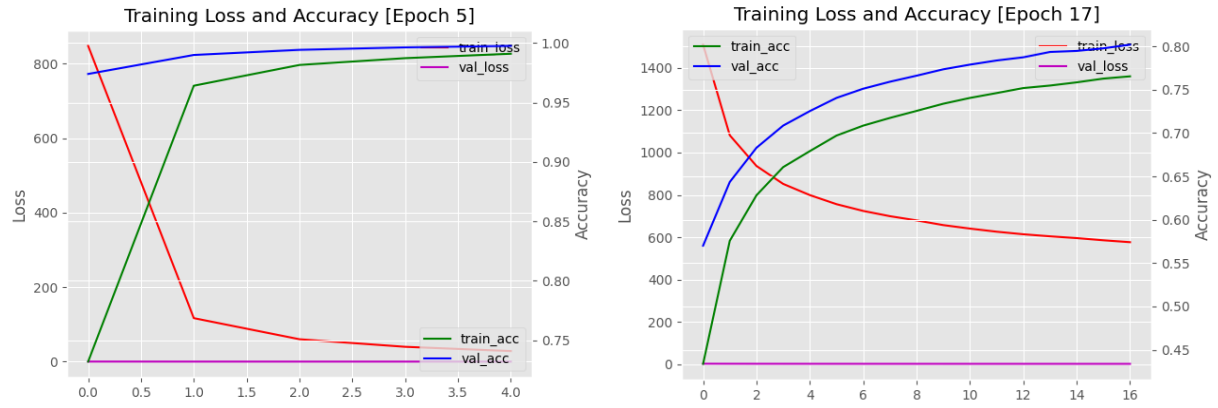


Figure 9: Inception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

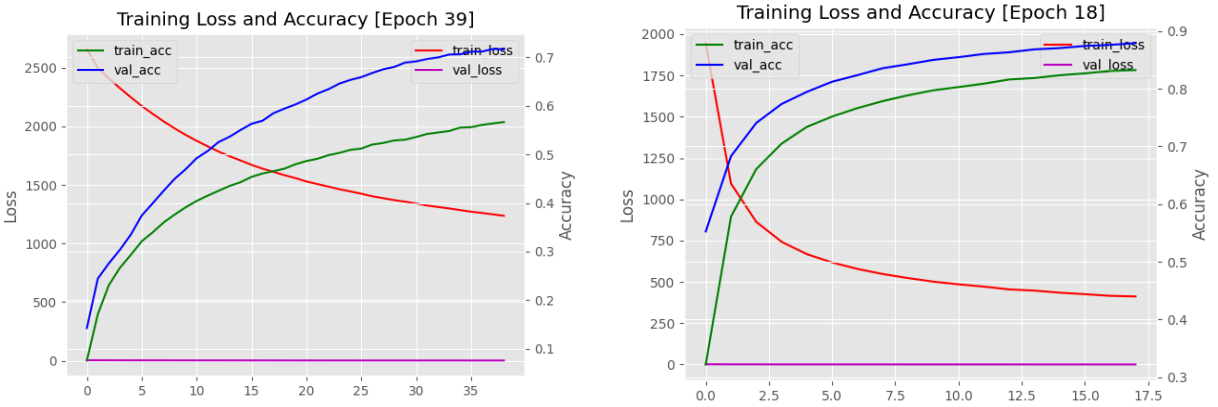


Figure 10: ResNet50 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

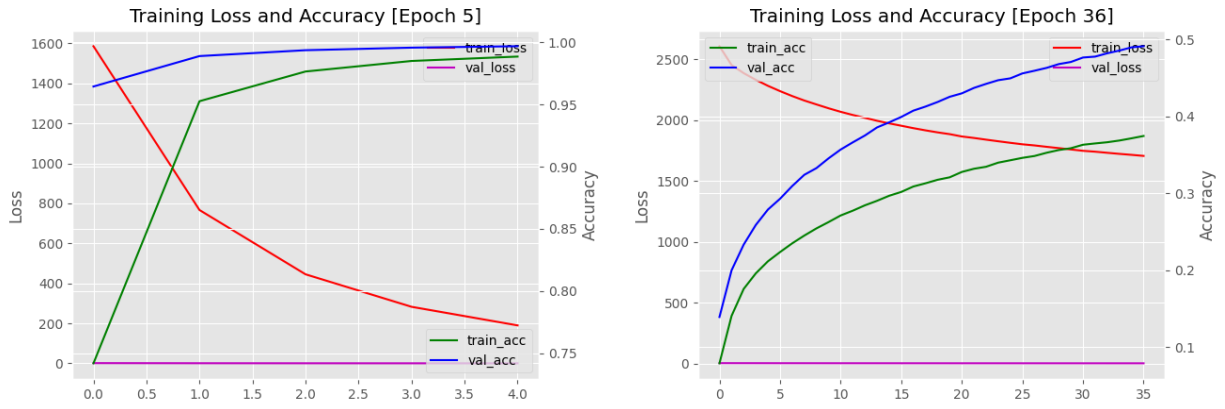


Figure 11: VGG16 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

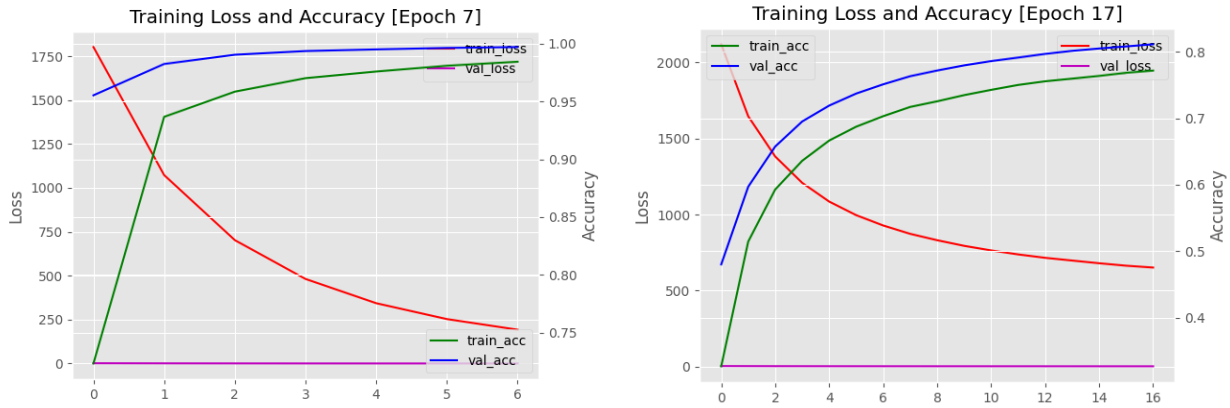


Figure 12: VGG19 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

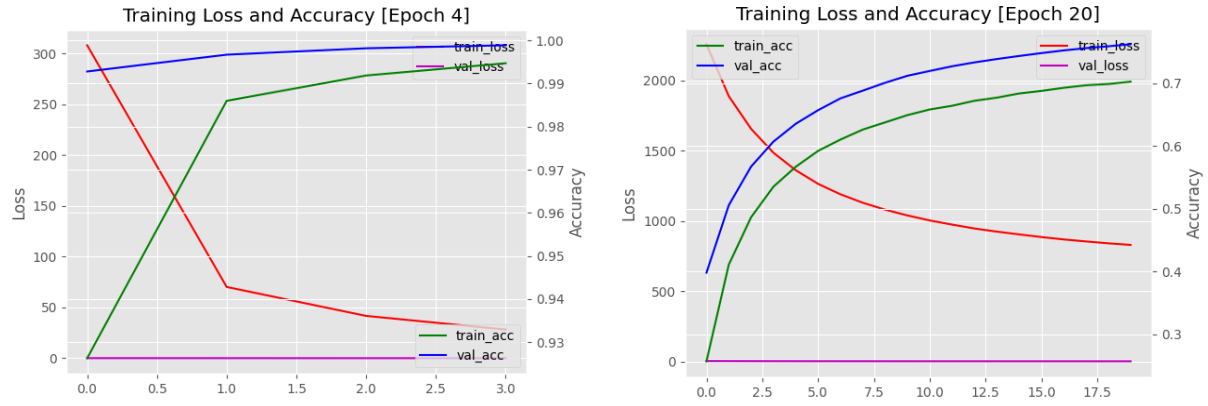


Figure 13: Xception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

For the ASL Alphabet dataset, the models that train best are Inception and VGG19. Both models trained to validation accuracies above 80% for both real and synthetic data, allowing for reliable comparisons between the two in further analysis. The Xception model trained adequately on synthetic data, but the validation accuracy under 80% is not ideal for reliable comparisons. The real training for ResNet50 and synthetic training for VGG16 both had poor training accuracy metrics as opposed to their counterparts, so these models will not be used for future analyses using the ASL Alphabet dataset since any comparisons between real and synthetic would be insignificant because of the gap in accuracy between real and synthetic data training.

The same comparisons between the real and synthetic training with the Sign Language for Numbers dataset are shown in Figures 14, 15, 16, 17 and 18. For the training with the real numbers dataset, ResNet50 and VGG19 failed to reach 80% validation accuracy, while the other three models reached a sufficient validation accuracy of 80% or higher. For training with the synthetic Sign Language for Numbers dataset, ResNet50 failed to train above 50% validation accuracy, while all other models reached 80% or higher.

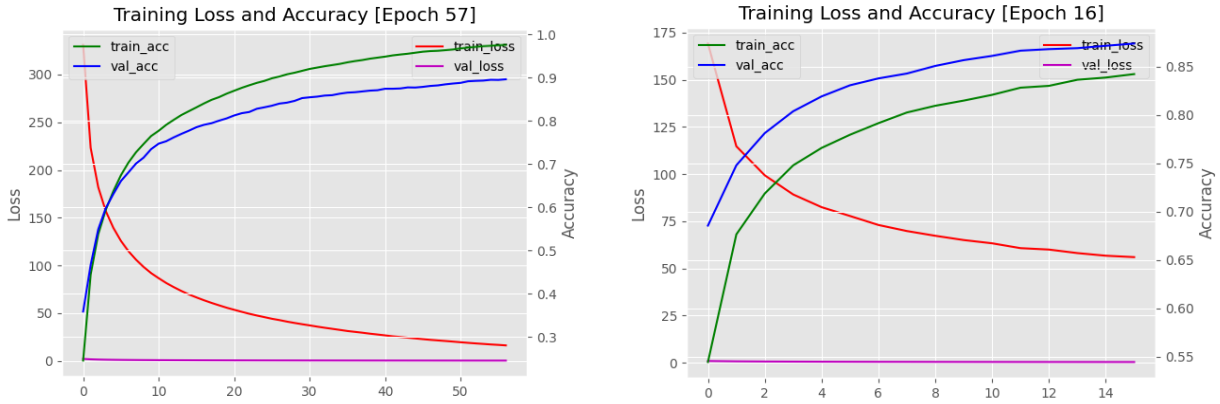


Figure 14: Inception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

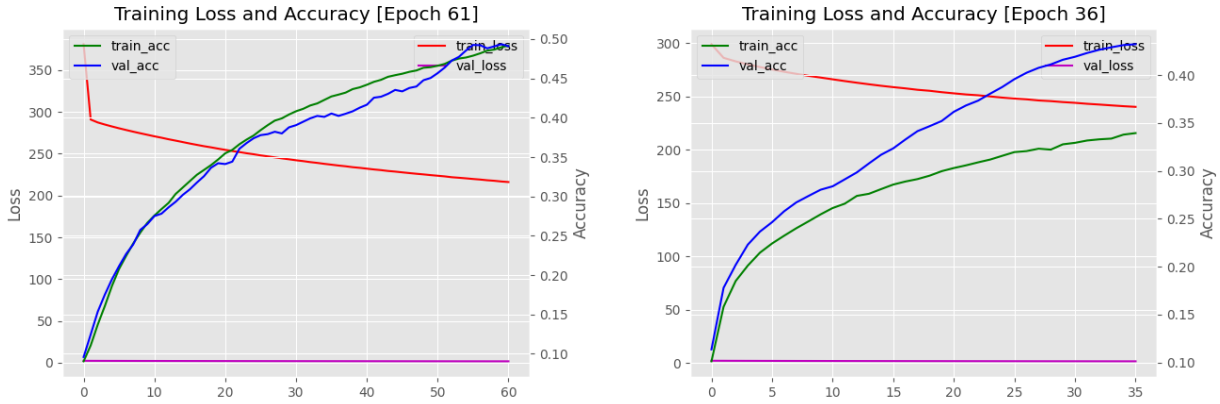


Figure 15: ResNet50 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

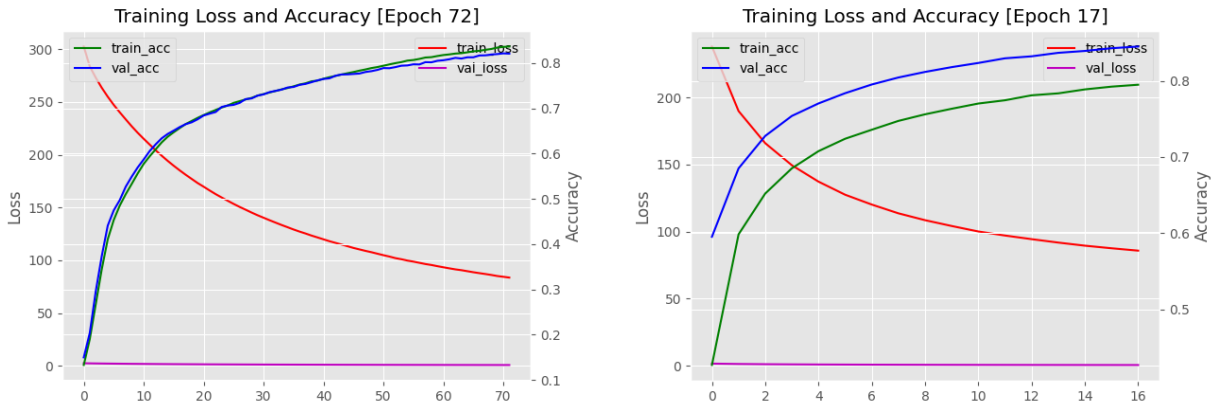


Figure 16: VGG16 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

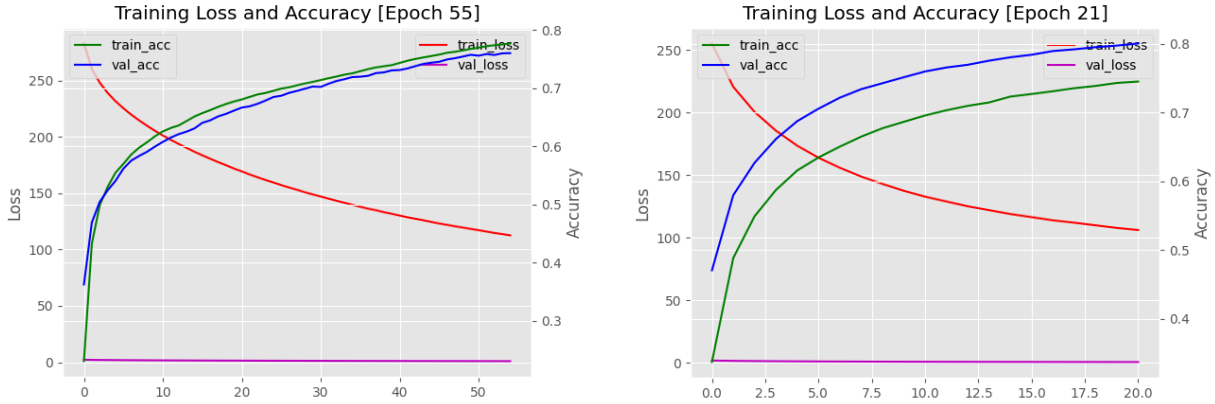


Figure 17: VGG19 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

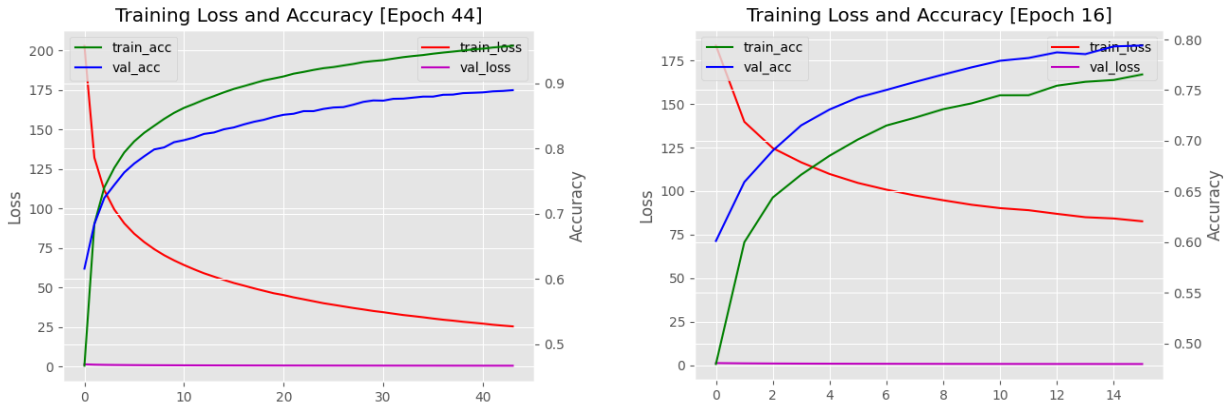


Figure 18: Xception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

For the Sign Language for Numbers dataset, the models that train best are Inception, VGG16, and Xception. These models trained to near or above 80% validation accuracy for both real and synthetic data, which allows for reliable comparisons in future analyses. The ResNet50 model struggled to train with both the real and synthetic data, which wouldn't allow for reliable comparisons of classification accuracy using this model. The VGG19 model trained well with the synthetic data but couldn't reach 80% validation accuracy when training on real data, which would also cause unreliable comparisons when using this model. For those reasons, these models will not be used for future analyses using the Sign Language for Numbers dataset.

2.2. Faster RCNN Gesture Object Detection

2.2.1 Overview

As an additional form of analysis for the generated synthetic images, object detection was performed using TensorFlow's object detection API. Object detection combines both localization, which is the drawing of a predicted bounding box around the identified objects within an image, and classification, which classifies the objects within the proposed regions from the localization. This form of analysis is more complex and can further prove any conclusions made using our synthetic hand gesture datasets.

For the data, the HANDS dataset was used, which contains full body images of 5 subjects performing 15 gestures, 12 being single-hand gestures and 3 being two-hand gestures. For our analysis, we stuck with single-hand gestures and got rid of the 3 two-hand gestures. Then, as mentioned before in the synthetic HANDS version 2 generation, the 'horiz' gesture was excluded because we did not have the time to implement the different arm positioning, so we stuck with 11 classes. This was done because the faster RCNN is trained using full images with ground truth annotations around the 'object' or, in our case, around the hand, which is used to train the region proposal network and localization of the model architecture. The matching synthetic dataset was also made with 5 virtual MakeHuman models to simulate the 5 subjects, but with 2,000 images per gesture.

The real and synthetic data, as well as combinations of the two, were used to train and test the faster RCNN. Some collected training metrics, as well as all the mAP (mean average precision) scores for each evaluation that was run are recorded below to validate the function of the faster RCNN.

2.2.2 Real and Synthetic HANDS Dataset Training Validation

For the faster RCNN config, a ResNet50 Region Proposal Network (RPN) pretrained on the MS-COCO dataset was used. The data input pipeline included data augmentation for random horizontal flips and random cropping to add additional image randomization. This was especially necessary for the synthetic images because all randomization within the Unity environment was removed to simplify the process of annotating the ground truth images. To match the synthetic datasets, the 'horiz' gesture was removed from the real HANDS dataset so the real and synthetic datasets could be directly compared. All training was done with a batch size of 1 image per step, using random horizontal flips and random image cropping for data augmentation. The exact configurations of the model can be seen in the configuration file. The total number of steps was adjusted depending on the number of images being used for training, generally ranging from 15,000 to 60,000 steps. The learning rate for all training starts at 0.003, drops to 0.0003 halfway through training, and towards the end, goes to 0.00003. This is done so

the model trains quickly early on, but as the model improves, the learning rate is reduced to prevent any instability in the training weights as the model gets closer to it's ideal weights for the training data.

Training was first done with the real HANDS dataset, training on subject 1 and evaluating on subject 2. Metrics from the training process can be seen below in figure 19, which shows the losses of the model. After 15,000 steps the total loss of the model was significantly reduced. Using the generated weight checkpoints of the model, an evaluation was run on subject 2. An example evaluation can be seen in figure 20, where the tested image from subject 2 was overlayed with the proposed regions and predicted class, along with the confidence value.

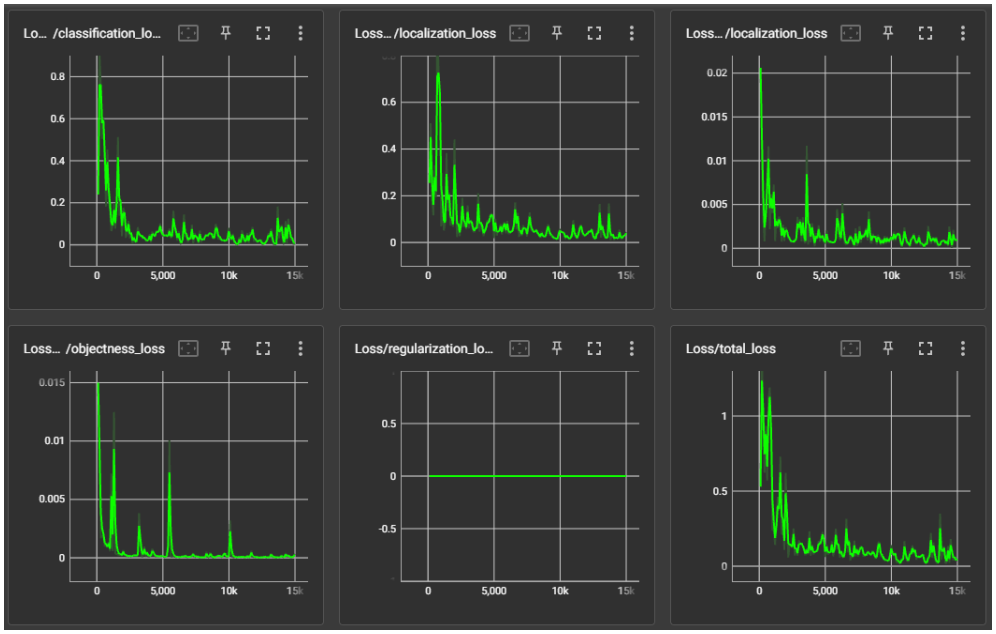


Figure 19: Loss Metrics While Training with Real HANDS Subject 1

Figure 20: Bounding Box and Classification Output Overlayed on Tested Real HANDS Subject 2 Image

After validating the real HANDS dataset, we then started training with the synthetic HANDSv2 dataset. Simulating the validation of the real data, the model was first trained on the synthetic model 1 and evaluated on synthetic model 2. After 50,000 steps, the total loss was very low, finishing around .02 total loss. The loss metrics can be seen in figure 21. The trained model was then evaluated on an image from the synthetic HANDSv2 dataset model 2 overlayed with the predicted bounding box and classification, as well as the confidence value, seen in figure 22.

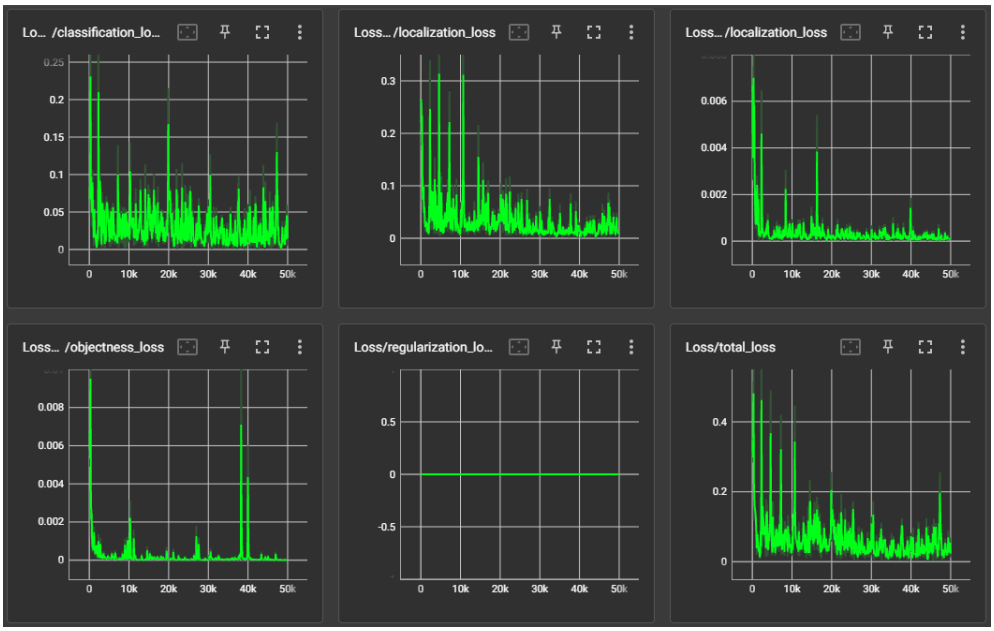


Figure 21: Loss Metrics While Training with Synthetic HANDSv2 Model 1

Figure 22: Bounding Box and Classification Output Overlayed on Tested Synthetic HANDSv2 Model 2 Image

Through these two tests, the training of the faster RCNN with both real and synthetic data was validated, as both models successfully localized and classified the gesture of a different subject or model.

2.2.3 Combinations of Real and Synthetic Training and Testing

After validating the ability to train the faster RCNN with the real and synthetic HANDS data, various combinations of data were used to train and test the model. The datasets for training and testing as well as the mAP score for each test are shown in table 5 below.

Table 5: mAP Score for Various Faster RCNN Tested Cases

Trained on	Tested on	mAP
Real subject 1	Real subject 1	1
Real subject 1	Real subject 2	0.58
Real subjects 1,3,4,5	Real subject 2	0.723
Synthetic model 1	Synthetic model 1	0.9895
Synthetic model 1	Synthetic model 2	0.8573
Synthetic models 1,3,4,5	Synthetic model 2	0.8483
Real subject 1	Synthetic model 1	0.2244
Synthetic model 1	Real subject 1	0.1228
Real subject 1 and synthetic model 1	Real subject 2	0.5706
Synthetic model then real subject 1 sequentially	Real subject 2	0.5747
Real subject 1 and all synthetic models	Real subject 1	0.7289
Real subject 1 and all synthetic models	Real subject 2	0.4311

Full testing of the real data was done, training with subject 1 and testing on itself and subject 2, as well as training with subjects 1, 3, 4, and 5, and testing on subject 2. From these results, it is evident that training with more of the real HANDS subjects helps the model generalize on unseen data. This can be seen in the mAP score from training with subject 1 and testing on subject 2, which was only 0.58, but after training with the other three subjects, the mAP score increased to 0.723. This is likely because each subject in the real HANDS dataset has one single background, so training with more of the subjects exposes the model to more diverse backgrounds, helping the model to differentiate the background from the gesture during evaluation. In the example output images from the evaluation, the detection from training with only subject 1 in figure 23 showed a misclassification on the subject's left hand. However, figure 24 shows the detection output after training with subjects 1, 3, 4, and 5, where both hands were classified correctly. Although this training introduced a new persistent false positive around the corkboard, the results still improved overall for the detection of the hand gestures.

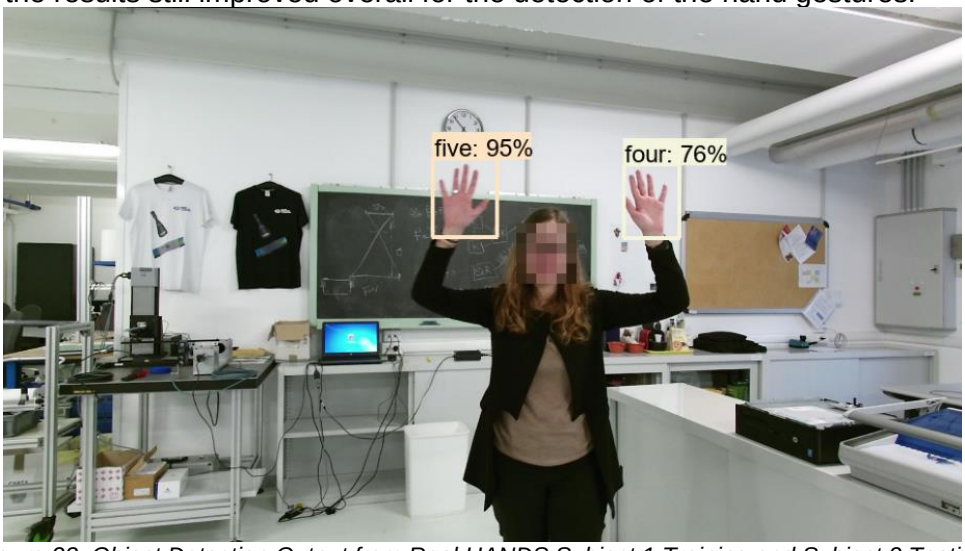


Figure 23: Object Detection Output from Real HANDS Subject 1 Training and Subject 2 Testing

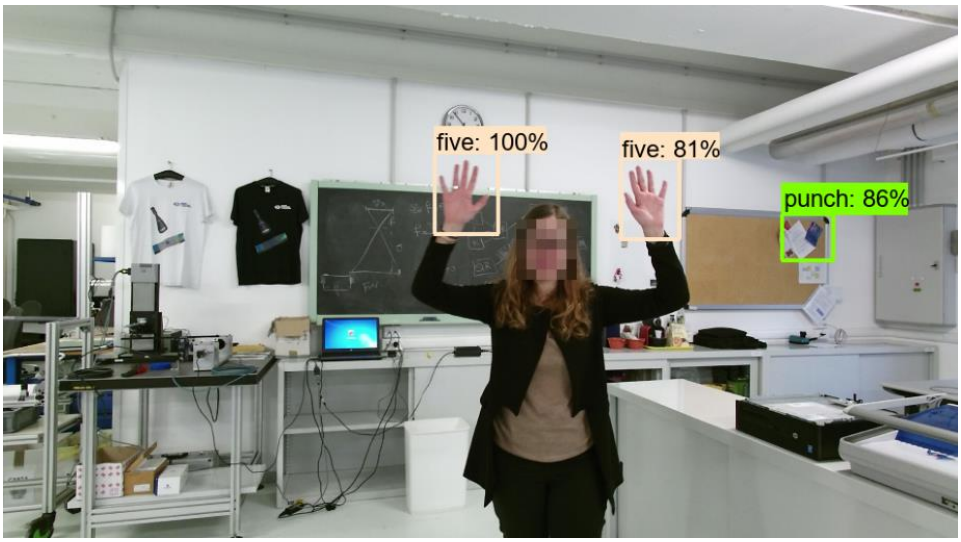


Figure 24: Object Detection Output from Real HANDS Subjects 1, 3, 4, and 5 and Subject 2 Testing

For the synthetic data, the model was trained on synthetic model 1 and tested on synthetic models 1 and 2, and was also trained on synthetic models 1, 3, 4, and 5 and tested on model 2. Just from training on the first synthetic model, the mAP when testing on model 2 was already very high, at 0.8573. Training with the remaining synthetic models resulted in an equivalent mAP score of 0.8483, which is likely because of both the lack of complexity in the images and the already diverse backgrounds. Since there was very little time left in the semester, the dataset was made with only one hand performing the gesture and with no randomization of gesture location. This made the collection of ground truth labels much easier, but when training with the model, it won't struggle as much with localizing, hence the higher mAP score. As for the lack of improvement between training with one or four synthetic human models, this is likely because the diverse backgrounds on one human model already helped the faster RCNN to differentiate the background from the gesture. Therefore, even after adding more models, the mAP doesn't improve. As seen in figure 25 and 26, training on one or four synthetic models showed practically no change in the detection output, which is largely due to the lack of complexity in the images.

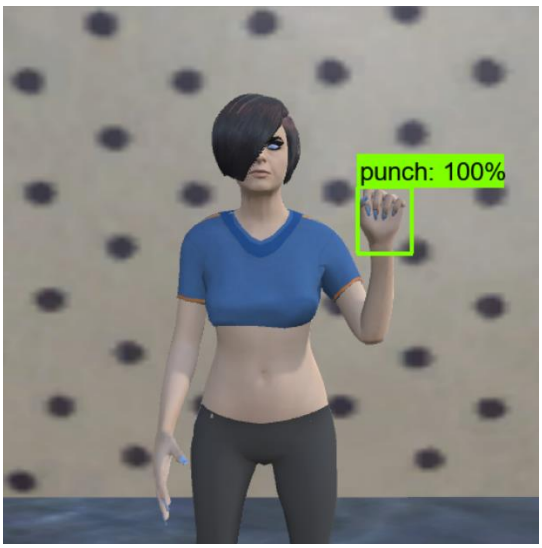


Figure 25: Object Detection Output from Synthetic HANDSv2 Model 1 Training and Model 2 Testing

Figure 26: Object Detection Output from Synthetic HANDSv2 Models 1, 3, 4, and 5 Training and Model 2 Testing

As for testing real and synthetic data together, the first case was trained with the real subject 1 and tested on the synthetic model 1, and vice versa. The mAP score for real on synthetic was 0.2244, and for synthetic on real, it was 0.1228. These low mAP scores were expected for various reasons. First, the synthetic images, even though they have a human model and diverse backgrounds, are still 'cartoony' and lack the detail of a real image. Another cause, as stated before, is the lack of complexity in the synthetic images. The real images have subjects that use both hands to perform the gesture as they move their hands around their bodies, while the synthetic images just have the models use one hand to perform the gesture without moving the arm at all. Finally, though the synthetic images have very diverse backgrounds, they lack depth. The subjects in the real images sometimes moved around in the environment, but in the synthetic images, the models stayed standing in one place. The background image for the synthetic data is also lacking in its simulation of a real situation, as well as any depth information. The background images are very close up pictures of random things like grass or a donut, but it doesn't really simulate any kind of depth like standing in a room, nor does it simulate the perspective of viewing objects in a room instead of looking closely at the textures of said objects. For the detection output of these cases, most tested images showed no drawn bounding boxes because the classification confidence failed to pass the set minimum score threshold of 0.3, so no figures are shown.

To potentially make up for the sim-to-real gap in our data, the model was trained with both the real and synthetic data and tested on subject 2. However, accuracy either remained the same or completely dropped. For the cases where the real subject 1 and synthetic model 1 were trained both together and sequentially (with the synthetic data first), the mAP scores were 0.5706 and 0.5747 respectively. These scores are effectively the same as the original mAP of 0.58 when testing subject 2 with the model trained on subject 1. When training with the real subject 1 and all the synthetic models, the mAP dropped significantly to 0.4311 when testing on subject 2, and 0.7289 when testing on subject 1. This emphasizes the previous notion that the synthetic data is too different than the real data to help with training the model. As more synthetic data is used along with real data, the training set is getting saturated with synthetic data and the model will begin to suffer when testing on real data. In figure 27, the model trained on both the real and synthetic data subject and model 1 only classified one hand correctly and gave a complete false positive, but it did at least localize the second hand. However, in figure 28 with the model trained on all the synthetic models and the first real subject, besides the single correct hand, the

other hand was completely missed and the clock in the background became a very persistent false positive at high confidence scores, resulting in a slightly lower overall mAP score.

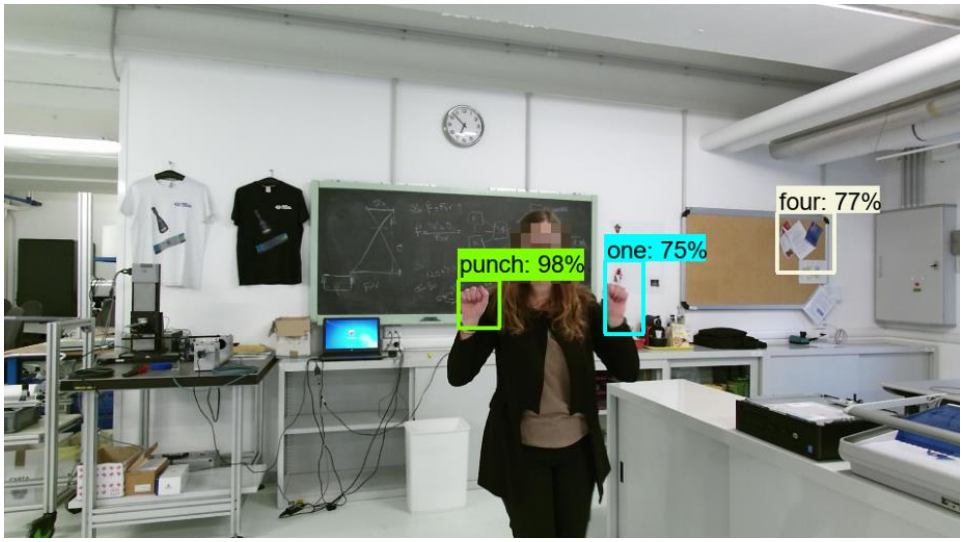


Figure 27: Object Detection Output from Synthetic HANDSv2 Model 1 and Real HANDS Subject 1 Training and Real HANDS Subject 2 Testing

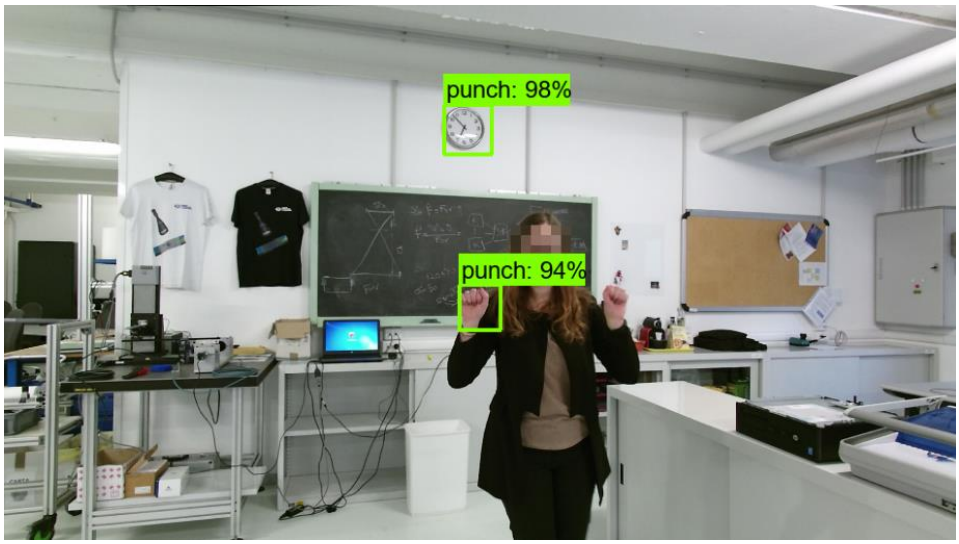


Figure 28: Object Detection Output from All Synthetic HANDSv2 Models and Real HANDS Subject 1 Training and Real HANDS Subject 2 Testing

2.2.4 Potential Improvements and Conclusion

For both the classification and the object detection, our synthetic data could successfully train models that could then be used to evaluate other synthetic data with accuracy similar to what is achieved with training and testing on real data. However, for the object detection, the synthetic data provided no improvement to the faster RCNN's ability to localize and classify gestures on real images. In fact, since the reality gap of our synthetic data is too significant, it reduces the accuracy of the faster RCNN. If we want to reduce the reality gap, we would need to make improvements to our dataset generation system. These improvements could be with regard to the background environment, using more realistic scenarios instead of random textures.

Another possible improvement could be to use a higher definition Unity render pipeline along with more detailed human models to more closely simulate the small details found in real images. With these improvements, it is theoretically possible to reduce the reality gap between the synthetic and real images, which could then help models generalize on real data better.

When considering training and testing on only synthetic data, the models for classification and object detection both train to very high accuracies. Knowing this, our dataset generation system could be used to create synthetic image datasets for classification or object detection within a virtual environment. The significance of these synthetic datasets for virtual computer vision continues to grow as the presence of virtual games and societies expands. With these synthetic datasets, training models for virtual classification and object detection would be extremely simple and would help to accelerate the advancement of machine learning in the virtual environment.

Hand Gesture Recognition

Samuel Oncken, Steven Claypool

ROBOTIC ARM TRAINING ENVIRONMENT REPORT

REVISION - 1
25 April 2023

ROBOTIC ARM TRAINING ENVIRONMENT REPORT
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72 4/25/2023
Author Date

APPROVED BY:

Samuel Oncken 4/25/2023
Project Leader Date

John Lusher, P.E. Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	4/25/2023	Samuel Oncken Steven Claypool		Release for 404 Final Report

Table of Contents

List of Tables

List of Figures

1. Robotic Arm Training Environment Description

1.1. System Overview and Purpose

The Robotic Arm Training Environment is a virtual system created in Unity to be used in the research of a graduate student, Pranav Dhulipala. This portion of our capstone is separate from the hand gesture dataset generation tool that was discussed throughout the report thus far. The goal is to train a real, full size robotic arm to slow down, stop, or move depending on the proximity of humans working around it. However, training this robot through real human interaction can be dangerous, as it would require humans to approach, touch, and engage with an untrained, large, moving object that is capable of injuring those around it. Instead of putting real humans at risk, Pranav's research presents the idea of training the robotic arm model virtually using Unity AI Gym and applying the virtually trained model to the real robotic arm structure. The next paragraph will provide a brief overview of how this virtual training environment works.

With the virtual robotic arm replica centered in the Unity training region, MakeHuman characters are spawned in various, random positions in the field and are animated according to randomly generated "behavior chains" (to be explained in a future section) to interact with the robotic arm. If a character gets too close to the robotic arm, they will immediately perform a "reaction" animation. All behavioral animations (Action, Reaction, etc.) are alterable by the environment coordinator to best depict real life interactions. The process of spawning a human model, performing a chain of random behavioral animations, interacting with the robotic arm structure, and removing the model from the scene is repeated according to the environmental coordinator's choice of training repetitions, which can be easily modified before the scene is run.

1.2. System Design Specifications

Before constructing the Unity environment, Pranav had several ideas that he planned to incorporate into the system functionality. The table below summarizes these design specifications and includes a current project status update on whether each function was properly implemented.

Table 1: Robotic Arm Training Environment Specifications

System Specifications:	Progress:
Use of "behavior" system, where a given behavior has randomized depictions (animations)	Met
Ability to chain behavior modules to form testing scenarios	Met
Ability for chains of behavior modules to play automatically for future training process	Met
Ability for characters to spawn/animate automatically at random in the testing field	Met
Once a character is spawned into the environment, they will be automatically positioned to be facing towards the robotic arm	Met

If a character collides with the robotic arm structure, they halt any animation and perform instead a “reaction” behavior	Met
System is easy to adjust/customize in terms of user controls and tests	Met
Robotic arm can be controlled as behavior chains are being performed	Met

1.3. Methodology

This section of the report will outline each important step in the Unity training environment development process and how it was implemented. The environment is controlled by a script called “AutomateMove” within the Human Controller game object placed into the Unity scene. The AutomateMove script contains all functionality described in the sections below. Screenshots of script segments will be periodically pasted into the report, but every line will not be included. The full AutomateMove script can be found in the GitHub repository along with all other project data.

1.3.1. Spawning, Positioning, and Removing of MakeHuman Models

Before any human animation or interaction can occur, a model must be present in the scene. The first step of the model spawning process is to locate each MakeHuman prefab model in the Unity assets folder. After all models (with common naming conventions) have been located, each prefab file is placed into the Human Models array, depicted publicly in the Inspector tab of the Human Controller object.

```
DirectoryInfo dir = new DirectoryInfo("Assets/NewAssets/Resources/Models/PrefabModels");       //selects directory to grab models from
FileInfo[] files = dir.GetFiles("mass*.prefab");                                              //places all files with name starting with mass and ending in .prefab
                                                                                              //from chosen directory into a files folder

foreach (FileInfo file in files)
{
    string name = file.Name.Split('.')[0];                                                    //for each file, disconnect the .prefab from the name
    fileList.Add(name);                                                                       //and add the model name into the list of file names
}
humanModels = fileList.ToArray();                                                             //transfers file of models into GameObject Array for use
```

Figure 1: Script Segment to Locate Usable MakeHuman Models

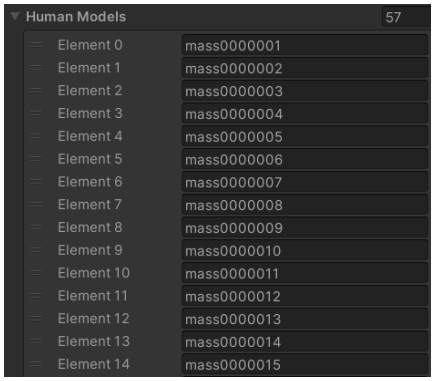


Figure 2: Human Models Array After Models are Added

To spawn a model into the scene, a prefab file must be collected from the Human Models array and a spawn position and rotation must be provided. Before physically instantiating the model into the scene, the position of the model is first generated, either at random or according to a fixed algorithm based on whether the environment is in Test Mode or not (Test Mode will be covered

in a future section). Assuming Test Mode is disabled, a random position in the training environment field is chosen. If this position overlaps with the robotic arm replica location, a new position is chosen. Depending on the randomly chosen position, a spawn angle is calculated using trigonometry to ensure that the model will be facing towards the center of the training region (towards the robotic arm). Slight randomness of ±15 degrees is added to the spawn angle to ensure that all models are not directly approaching the model from head-on angles.

```
float randX = Random.Range(-80f, 80f);                          //choose random location in XZ plane to spawn character
float randZ = Random.Range(-80f, 80f);                          //Y value doesnt matter, just value close to 0 so player will be walking on ground
while ((randX <= 25 && randX >= -25) && (randZ <= 20 && randZ >= -20))
{
    randX = Random.Range(-80f, 80f);                            //if model is spawned inside of arm region, repeat spawn location randomization
    randZ = Random.Range(-80f, 80f);
}
if (randZ == 0 && randX > 0)                                    //then depending on spawn location, calculate rotation necessary for character to face
{                                                               //the center of the map
    theta = -90;
}
else if (randZ == 0 && randX < 0)
{
    theta = 90;
}
else if (randZ > 0 && randX == 0)
{
    theta = 180;
}
else if (randZ < 0 && randX == 0)
{
    theta = 0;
}
else if (((randZ < 0) && (randX > 0)) || ((randZ < 0) && (randX < 0)))
{
    theta = Mathf.Atan(randX / randZ) * Mathf.Rad2Deg;
}
```

Figure 3: Script Segment Depicting Portion of Model Position and Rotation Calculation

After the spawn position and rotation is determined, a character can be instantiated into the scene. The character model is then added to an empty “spawned” array to indicate that a model is present in the scene. An Animator component is then added to the model and an Animation Controller (discussed more in next section) is selected to control the Animator. After a character has performed all behavior animations in a generated behavior chain, they are removed from the scene by destroying the model from the “spawned” array. The process continues by checking to make sure that the “spawned” array is empty before instantiating a new model into the scene.

```
string filename = $"Models/PrefabModels/{humanModels[HumanIndex]}";             //gets human model filename depending on randomly chosen index
GameObject spawnedModel = Instantiate(Resources.Load<GameObject>(filename), spawnPos, spawnRot);    //places chosen model in scene at random location and rotation
spawned[0] = spawnedModel;                                                      //indicates a new model is spawned
animatorBody = spawnedModel.GetComponent<Animator>();                           //gets Animator component on model
animatorBody.runtimeAnimatorController = controller;                            //places user-selected Animation Controller on Player Animator component
```

Figure 4: Script Segment Depicting Model Instantiation and Preparation for Animation

1.3.2. Behavioral Animation

As mentioned in the Overview and System Specifications, this system is “behavior” based. In this research, behavior is defined as a distinct type of motion/animation. The behaviors implemented in this project include Idle, Action, Reaction, Movement (both forward and backward) and Rotation (left and right). Each behavior has a unique set of animations that are used to depict it. These animations can be changed as the environment coordinator chooses. Currently, the Idle, Forward Movement, Backward Movement, Left Turn, and Right Turn behaviors each have a singular animation depiction. On the other hand, the Action and Reaction (both forward and backward reactions) each have 5 animation depiction options that are randomly chosen during the running of the Unity environment. The Animation Controller, shown in Figure 5 below, demonstrates how each behavior is its own state in a state machine. It is within the Animation Controller that a coordinator can alter which animation clips are used as well as how many potential depictions of a behavior are possible. Figure 6 shows each of the potential animation clips used to depict the Forward and Backward Reaction behaviors.

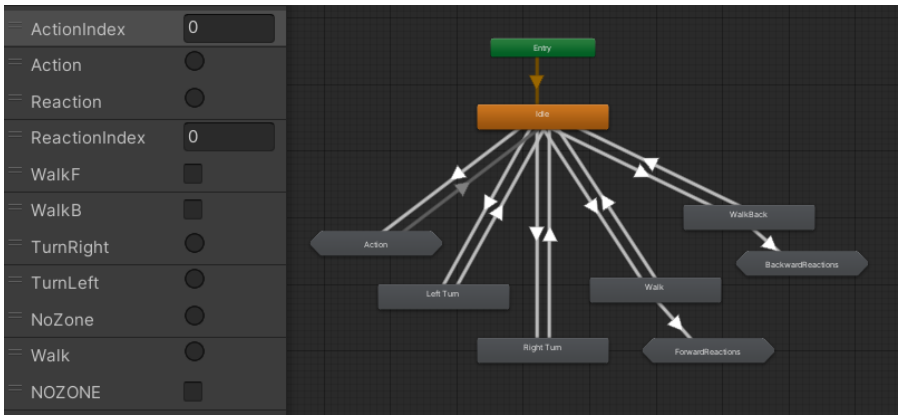


Figure 5: Animation Controller Used in Training System

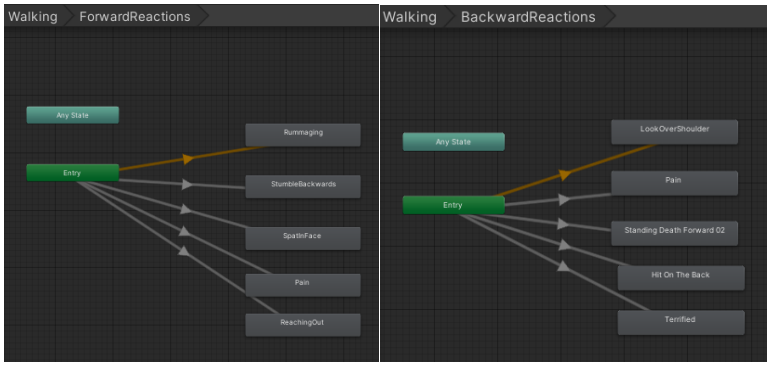


Figure 6: Potential Animation Depictions of Forward and Backward Reaction Behaviors

The Animation Controller is responsible for not only the performance of a singular behavior, but it also must be able to chain behavior performances through transitions. Each line between states represents a transition and is triggered by parameters on the left side of Figure 5. The parameters of the Animation Controller are modified by the AutomateMove script depending on the randomly generated behavior chains. Before spawning any models, the “BuildBehaviorChains” method is called within the AutomateMove script. This method considers 2 user-selected parameters. The first parameter is the number of behavior chains the coordinator wishes to train with. This parameter is similar to an iteration selection, as the number of behavior chains generated corresponds to the number of MakeHuman models being spawned, animated, and removed from the scene to help train the robotic model. The next parameter considered is the maximum length the coordinator wishes the behavior chains to be. If the coordinator selects that the maximum chain length is 3, then a behavior chain of “Action, Left Turn, Right Turn” is possible, but a behavior chain of “Action, Action, Forward Walk, Backward Walk” is not because that is one too many behaviors in the chain. For simplicity’s sake, behaviors are represented by characters. “a” represents Action, “F” represents Forward Walk, “B” represents Backward Walk, “L” represents Left Turn, and “R” represents Right Turn. The Reaction behavior is not able to be selected in the behavior chain generation process because the only instance that a Reaction animation should play is after a character interacts physically with the robotic arm region. After both parameters are inputted by the coordinator, the BuildBehaviorChains method randomly generates all behavior chains, indicating the end of a behavior chain with a period and storing each in the Behavior Chains array publicly viewable in the Human Controller Inspector window and shown in Figure 7. Figure 8 shows the BuildBehaviorChains method within the AutomateMove script.

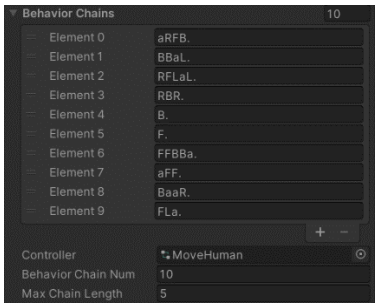


Figure 7: Example Behavior Chains Generated With Chain Num=10 and Max Chain Length=5

```
//this method is used to automatically create behavior chains
1 reference
void BuildBehaviorChains()
{
    string BehaviorChoices = "FBRLa";                       //string of potential behavior choices (character selected is behavior choice)
    int numChains = BehaviorChainNum;                       //use user-inputted number of behavior chains to build correct number
    BehaviorChains = new string[numChains];                 //create BehaviorChains array of size numChains (desired by user)
    for (int i=0; i<numChains; i++)                         //loop until numChains amount of behaviorChains have been created
    {
        string chain = "";                                  //create empty string for a chain
        int chainLength = Random.Range(1, MaxChainLength+1);//choose random length of chain from 1 to user selected number
        for (int j=1; j<=chainLength+1; j++)                //loop until chain contains the chainLength number of behaviors
        {
            if (j < chainLength+1)                          //if we are not at the last behavior on the chain,
            {
                int index = Random.Range(0, BehaviorChoices.Length);    //choose random index to pick a behavior
                chain += BehaviorChoices[index];                        //add behavior choice to initially empty chain string
            }
            else
            {
                chain += ".";                               //if the chain is on its last character, place a '.'
            }
        }
        BehaviorChains[i] = chain;                          //add the created chain to the BehaviorChains array
    }
}
```

Figure 8: BuildBehaviorChains Method Within AutomateMove Script

After all behavior chains are generated and the first model is spawned into the scene, a loop is entered that will not exit until all behavior chains are fully performed. During this loop, each behavior chain (represented as a string of characters) is parsed and depending on the character read during each parsing iteration, a parameter in the Animation Controller is altered to trigger a transition between behavior states. For instance, if the first character read in a behavior chain is “a”, then a random choice of Action animation depictions will be chosen using a random integer generator and the Action trigger will be enabled, beginning the Action animation performance. A variable corresponding to a time delay will also be set depending on the animation selected. If the Action animation selected to be played is 1.8 seconds in duration, then the loop needs to be paused by at least 1.8 seconds before continuing to trigger the next behavior state to be performed. This timing delay was the most challenging portion of this environment creation. It was also the most important. Without any delays, the loop would be carried out as fast as possible by the coordinators computer and as a result, all behavior triggers would be activated at the same time, resulting in animation overlap and poor quality of training material. To use time delays, the loop must be within a co-routine so that a specific function called WaitForSeconds can be used. After each trigger for a specific behavior was activated, the WaitForSeconds function was called with the time-delay entered as a parameter. Figure 9 depicts how time delay is implemented in the looping process.

```
for (int i=0; i<array_length; i++)                          //loop to read all BehaviorChains indexes
{
    animatorBody.SetBool("NOZONE", false);                  //reset boolean parameters in Animator to false
    animatorBody.SetBool("WalkF", false);
    animatorBody.SetBool("WalkB", false);
    while (behavior != '.')                                 //loop reading each index of behaviorChain inside of BehaviorChains array
    {
        if (behavior == 'a')                                //if 'a' is read, action is performed
        {
            float playTime = PlayAction();
            yield return new WaitForSeconds(playTime);      //must delay while loop run until action is fully performed to avoid overlap of animations
        }
        else if (behavior == 'B')                           //if 'B' is read, Walking Backwards animation is played
        {
            WalkBOn();                                      //calls for walking backwards boolean to be true
            yield return new WaitForSecondsRealtime(5f);    //waits 5 time units
            WalkBOff();                                     //turns walking backwards boolean to false (animation stops)
            yield return new WaitForSecondsRealtime(2.5f);  //delays to ensure animation is fully stopped before starting next one
        }
```

Figure 9: Loop Iterating Through all Behavior Chains with Time Delay

Moving back to the Action behavior example, after the Action trigger is enabled, a time-delay is determined based on which action animation was selected. If the action animation is 4.5 seconds in duration, then a time-delay of 5 seconds is implemented (slightly higher to be sure that no animation overlap occurs). The same methodology is used for Right Turn, Left Turn, Movement, and Reaction animations to ensure that the entire animation is performed without interruption.

```
1 reference
void WalkBOn() { animatorBody.SetBool("WalkB", true); }          //turns boolean for walking backwards animation to occur on in Animation Controller
1 reference
void WalkBOff() { animatorBody.SetBool("WalkB", false); }        //turns boolean for walking backwards animation to occur off in Animation Controller

1 reference
float PlayAction()
{
    float neededTime = 0;                                        //neededTime is decided based on duration of animation clip selected to play out
    int index = Random.Range(0, numActionAnims);                 //chooses random index from 0 -> amount of action animations-1
    animatorBody.SetInteger("ActionIndex", index);               //changes integer in Animation Controller to properly make a transition between animations
    animatorBody.SetTrigger("Action");                           //sets trigger for Action so transition can occur
    if (index == 0 | index == 2) { neededTime = 2f; }            //these are manually placed times based on duration of animation
    else if (index == 1 | index == 3) { neededTime = 5f; }
    else if (index == 4) { neededTime = 4f; }
    return neededTime;                                           //returns neededTime useful in delaying Start method
}
```

Figure 10: Example Animation Controller Parameter Triggering and Time-Delay Determination

Now, the process of triggering a Reaction animation will be described. As mentioned previously, the Reaction behavior is dependent on a model colliding with the robotic arm replica. An invisible, cylindrical game object closely surrounding the robot replica was created to process this collision, shown in Figure 11.

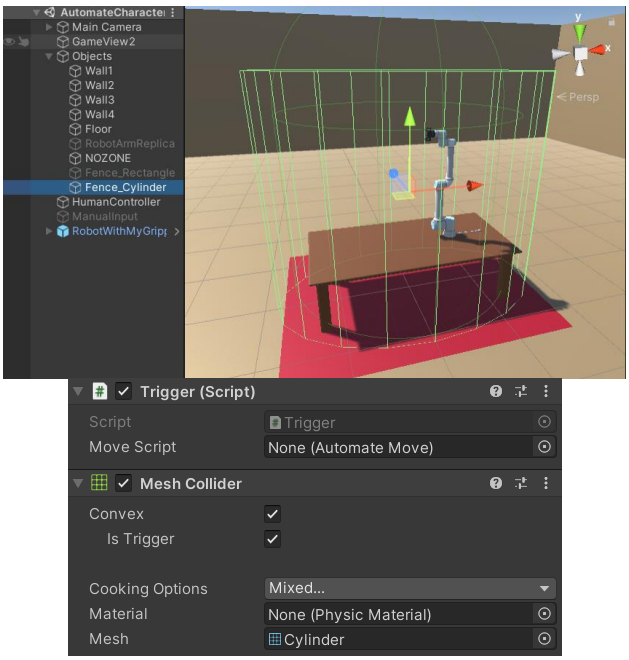


Figure 11: Fence Trigger Region Around Robotic Arm with Attached Collider and Trigger Script

The cylindrical “Fence” object contains a mesh collider with IsTrigger mode enabled. This means that instead of acting like a normal collider which would act similarly to a wall, the collider can be used to trigger an entry condition. Using this property, a separate script was developed called “Trigger” which was then placed on the Fence component. The purpose of the Trigger script is to act as an alarm, warning the main AutomateMove script that the Fence collider has been breached and that a Reaction animation should play. The Trigger script is fairly straightforward. First, the AutomateMove script must be located so that the 2 scripts can communicate. This is done by tagging the Human Controller game object (which contains the AutomateMove script) with the name “GameController”. The Trigger script is then able to parse all game objects in the

scene and find the one with the GameController tag. From there, the AutomateMove script can be accessed and communicated with. The OnTriggerEnter method is entered when the Fence collider (with IsTrigger mode enabled) detects that it has been breached. After the breach, a Boolean in the AutomateMove script is set to true which tells the system that a Reaction animation should play. Figure 12 shows the full Trigger script.

```
Unity Script (2 asset references) | 0 references
public class Trigger : MonoBehaviour
{
    public AutomateMove moveScript;            //uses AutomateMove script

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        GameObject g = GameObject.FindGameObjectWithTag("GameController");          //finds object with GameController tag which houses AutomateMove script
        moveScript = g.GetComponent<AutomateMove> ();                               //gets the script component from that object with tag
    }

    Unity Message | 0 references
    void OnTriggerEnter(Collider other)
    {
        moveScript.trigger = true;                          //when Fence trigger is set off, change boolean value trigger inside of AutomateMove script to true
    }

}
```

Figure 12: Trigger Script

Back in the AutomateMove script, the Update function (which is called every frame) is constantly checking the reaction trigger Boolean to see if the Fence has been breached. Once the Trigger script sets the reaction trigger Boolean to true, the Update function immediately catches the change and calls the “PlayReaction” co-routine. Inside PlayReaction, the system checks which behavior had been performed prior to the breach, which can either be a forward walk or a backwards walk. Depending on how the player entered the robotic arm region, a random reaction animation is selected to play and once again a delay is implemented. The Reaction animation takes precedence over any previously playing animation clips. After the Reaction is performed, the rest of the behaviors in the behavior chain are ignored, the character is removed from the scene, and the next model is spawned to begin a new chain of behaviors. Figures 13 and 14 show the code implementation within the AutomateMove script.

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    //we need to constantly check if player has interacted with the Fence collider which has its own Trigger script connected to this script
    if (trigger == true)                                    //every frame checks trigger (which is changed to true when interaction occrs)
    {
        trigger = false;                                    //immediately change value of trigger back to false so that update is not repeatedly called
        animatorBody.SetBool("NOZONE", true);               //set boolean parameter in animation controller to indicate that player is in Fenced region
        StartCoroutine(PlayReaction());                     //play a reaction animation
        ReactAnimDone = true;                               //indicate that a reaction has been played
    }
}
```

Figure 13: Update Function Checking for Fence Breach and Calling Reaction Behavior

```
IEnumerator PlayReaction()
{
    int index = 0;
    if (behavior == 'F')                                    //if player entered walking forwards,
    {
        index = Random.Range(0, numFReactionAnims);         //use number of forward reaction animations
    }
    else if (behavior == 'B')
    {
        index = Random.Range(0, numBReactionAnims);         //same logic for if player entered backwards
    }
    animatorBody.SetInteger("ReactionIndex", index);        //Random Reaction animation is chosen
    yield return new WaitForSecondsRealtime(1.5f);          //pause running so reaction animation can be fully played out
}
```

Figure 14: PlayReaction Co-Routine

1.3.3. Test Mode

Incorporating a Test Mode into the training environment is vital for users to be able to test new animation clips, manually test behavior chain transitions and validate environment functionality. The goal of Test Mode is to make the environment as customizable as possible by the user with

little to no automation when enabled. To implement this, a TestMode Boolean was created in the AutomateMove script that would disregard certain automation functions such as behavior chain generation and model spawn condition randomization. With TestMode enabled, new user-selected parameters would become vital to the running of the environment. TestMode must be enabled before a user can select a test type. Currently, 3 test types have been implemented. The first test type is a manual spawn test. In this test, a user must manually generate their own behavior chains and indicate the precise spawn location and rotation parameters for a character to be instantiated. The next 2 test types are customizable validation tests. These tests are called the Multi Angle Reaction Test and the Reaction Distance Test. More on these tests will be included in the Validation section of this paper.

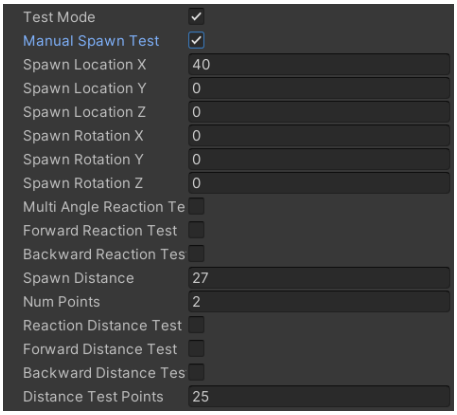


Figure 15: Test Mode Customizations

Rigorous Test Mode specific user-input checks have been incorporated into the AutomateMove script. The system checks that if TestMode is enabled, a test type is also enabled. The system also checks that if a test type has both forward and backward test options, one must be selected for the environment to run. In addition, the system checks that no more than one test type or forward/backward option is selected at a time. For each test type, it was mentioned that new user-selected parameters become vital to the Unity run. In the manual spawn test mode, the system checks that the spawn position is not outside of the training region nor on top of the robotic arm structure. In the Multi Angle Reaction Test, the system checks that the number of spawn points is positive and the spawn distance from the center of the map is 25 units or above (to ensure the character has room to walk before entering robotic arm region). In the Reaction Distance Test, the system checks that the number of test points is positive and that the starting distance is not too close to the robot. If any of these checks fail, the Unity environment exits play mode, and a console output message helps the user determine what caused the failure.

```
if (TestMode == false)
{
    BuildBehaviorChains();              //if TestMode is not on, enable automatic creation of behavior chains and carry on
}
else
{
    //The following are edge-case checks to ensure proper user entries
    //if test mode is on, a test type needs to be selected. Check for this before continuing.
    if (MultiAngleReactionTest == false && ManualSpawnTest == false && ReactionDistanceTest == false)
    {
        EditorApplication.isPlaying = false;
        Debug.Log("Please select a test type. Manual Spawn Test, Multi Angle Reaction Test, or Reaction Distance Test.");
    }
    //if a multi angle reaction test is selected, a user must designate forward or backward type
    if (MultiAngleReactionTest == true && ForwardReactionTest == false && BackwardReactionTest == false)
    {
        EditorApplication.isPlaying = false;
        Debug.Log("While running a Multi Angle Reaction Test, you must select between the forward or backward test options.");
    }
    //if a reaction distance test is selected, a user must designate forward or backward type
```

Figure 16: Script Segment Showing Some User-Input System Checks

1.4. System Validation

To validate this system, it was necessary to ensure that all previously discussed environmental specifications had been met and all methodologies functioned as desired. Table 2 summarizes the validation tests discussed in the next few paragraphs of this section of the paper. With Test Mode disabled, it was validated that the system accurately generated the number of behavior chains that were desired by the user. In addition, it was ensured that each behavior chain did not exceed the maximum chain length set by the user before the Unity run. Each automatically generated behavior chain was validated to be constructed properly, being a string of behaviors represented by characters. Each character (behavior) in the behavior chain is accurately interpreted and depicted in the form of an animation during the running of the environment. When a behavior (such as Action or Reaction) has several potential depictions, it was validated that all choices of animation clips included in the Animation Controller for the specific behavior can be picked and performed at random. During the system run, it was validated that characters could not be spawned within the robotic arm region. It was also validated that every behavior within a behavior chain is fully performed before transitioning to the next behavior in the chain. After all behavior chains were read and performed, the system automatically stops the Unity run. In addition, if any user inputs such as behavior chain number or max chain length were detected to be invalid (such as a negative number), it was validated that the system will not begin running and would instead output a console message to help the user determine what caused the issue.

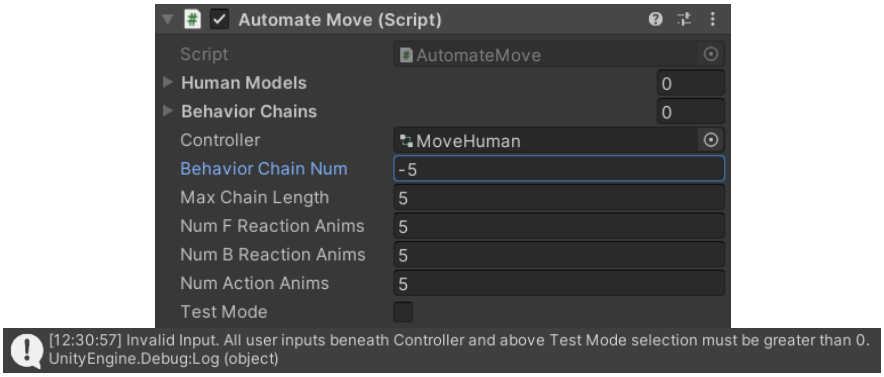


Figure 17: Example Invalid User Input with Helpful Console Message

Using Test Mode, it was possible to perform extra, customizable validation test scenarios that could extensively test the functionality of the system. The first validation test created was the Multi Angle Reaction Test. The Multi Angle Reaction Test spawns MakeHuman models in a circle around the robotic arm region and validates that at any angle of approach, a reaction animation will be performed if the character collides with the robotic arm region. In this test, a system coordinator must input a spawn distance (radius from the center of the training region) and the number of test points to be used. Using the user-selected number of test points, the system calculates the angle shift to spawn each character iteratively. For example, if the user wants 5 test points, the system will calculate that a character must be shifted around the circle by 72 degrees at each iteration of spawning (360/5). This angle shift is then translated into spawn coordinates using the radius selected by the user. Finally, depending on whether the test is in forward or backward mode, the character spawn rotation is determined to make sure that the character either faces directly toward or directly away from the model.

```
//if a MultiAngleReactionTest is being run, print console message and move character spawn to match desired next angle
if (TestMode == true && MultiAngleReactionTest == true)
{
    Debug.Log("Iteration #"+(iteration+1)+" complete. SpawnAngle = " + (iteration * angleShift*Mathf.Rad2Deg));
    iteration += 1;
    addAngle = iteration * angleShift;
    spawnX = spawnDistance * Mathf.Cos(addAngle);
    spawnZ = spawnDistance * Mathf.Sin(addAngle);
    yield return new WaitForSeconds(.5f);
}
```

Figure 18: Code Segment from Multi Angle Reaction Test Logic

It was validated that with 50 points (angle shift of 7.2 degrees per iteration), every time a character encountered the robotic arm region (whether it was forward or backward approach), a randomized reaction animation would consistently be performed. This validates that no matter how the character approaches the region, the system will properly interpret the collision and transition the character animation controller to perform a reaction.

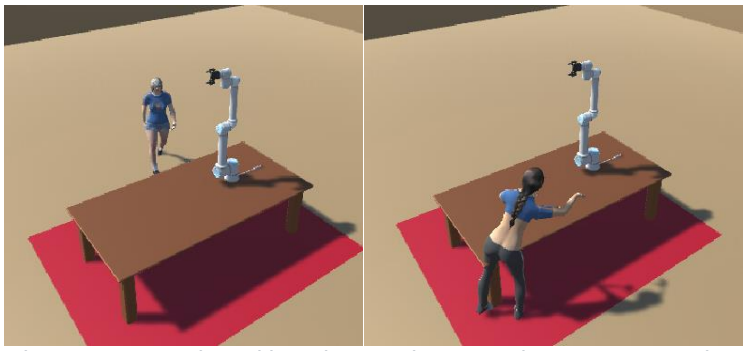


Figure 19: Example Multi Angle Reaction Test Character Approaches

The next validation test is the Reaction Distance Test. The purpose of this test is to ensure that at any stage of a walking animation (whether it just began or is close to completion), a reaction will always be performed at the instant the character interacts with the robotic arm region. In this test, a character is spawned on one axis at a starting distance away from the model. The system coordinator must select a starting spawn distance and the number of test points to use. The test begins by spawning a MakeHuman model at the starting spawn distance in the z direction and animating the character to walk forward or backward (depending on the test type) until they make contact with the robotic arm region. After the character makes contact, the system coordinator ensures that a reaction animation was properly performed. After the reaction performance, the system destroys the spawned model and instantiates a new one at (starting distance + 2*(iteration#)). For example, if the starting distance set by the user is 60, the character will first approach the model from 60 units away in the z direction. After the first iteration, the character will be destroyed and a new one will spawn at 62 units away. This process repeats until the number of test points is met. If the user selected that the number of test points is 3 for instance, the test would perform 3 iterations starting at 60, moving to 62, then finally performing the last iteration at 64 units away from the center of the training region.

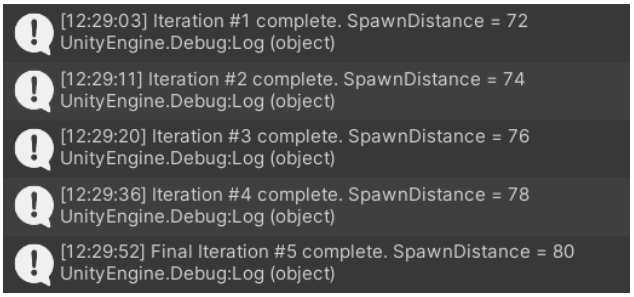


Figure 20: Example Spawn Distance Test Console Output with Start Distance=72 and Test Points=5

This validation test was extremely helpful in finding a slight hole in the system that had to be fixed. It was observed that when a character approached the model from a far spawn distance (where it would take the entire walking animation time for the character to be close to the robotic arm region), the character would not perform a reaction, but would still be destroyed for the next character to be spawned. This issue showed that if a character contacted the robotic arm at a late stage of the walking animation, the reaction would be ignored. It was determined that this issue stemmed back to the Animation Controller. At the end of the walking animation, an animation transition had been implemented so that the character would slow its movement and come to a gradual stop, rather than abruptly changing from the walking state to the idle state. However, the problem with this was that while transitioning back to idle, the character was still technically walking and therefore could interact with the “Fence” trigger. But since a transition cannot occur from the idle state to the reaction state, the reaction animation could not be called to play. This problem was solved by reducing the animation transition time between the movement and idle states to ensure that the character does not interact with the “Fence” collider while exiting the movement behavior state.

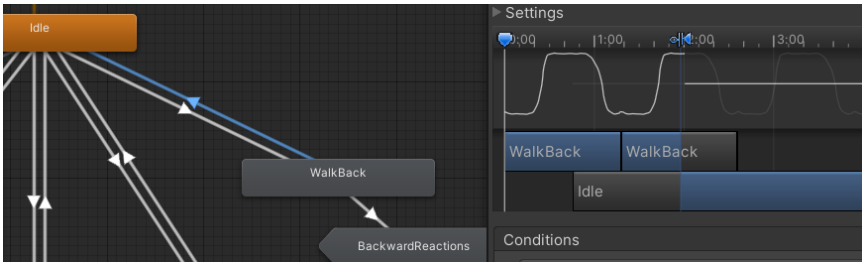


Figure 21: Animation Transition Solution to Issue Within Animation Controller

After this solution was implemented, the problem was eliminated and after a full run of the Reaction Distance Test, it was validated that a reaction animation would always be performed no matter what how soon after the walking animation begun or how close the walking animation was to ending. This test was performed for both forward and backward movements.

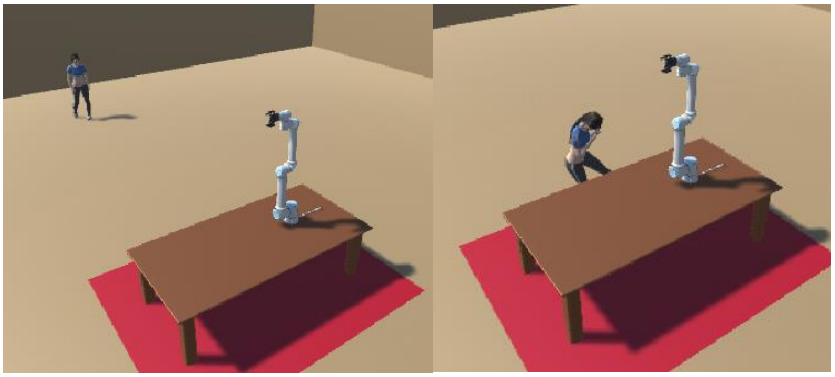


Figure 22: Example Spawn Distance Test from Far Starting Point

Finally, the last validation test involved confirming that the robotic arm was fully functional while the training environment was running. This was validated by disabling Test Mode and beginning the Unity run with the robotic arm prefab included in the scene and the Manual Input script enabled. While characters were being spawned and animated, keyboard controls were confirmed to affect the movement of the robotic arm and no interferences/malfunctions were observed.

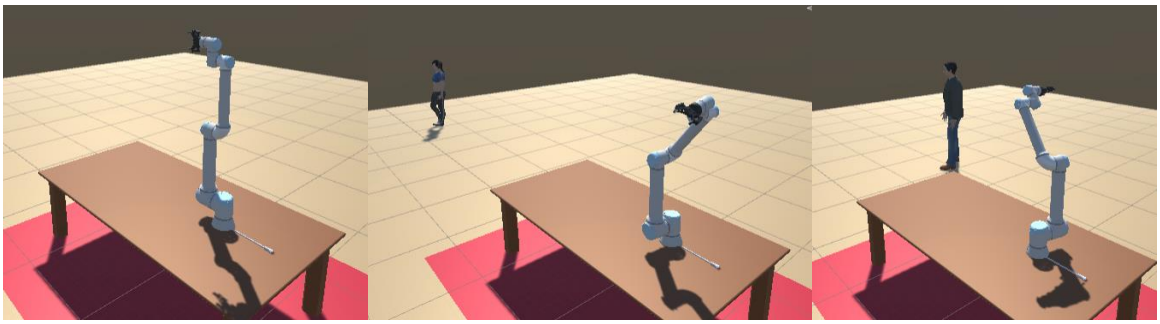


Figure 23: Movement of Robotic Arm While Training Environment Runs

Table 2: Summary of Validation Tests for the Robotic Arm Training Environment

Validation Test	Result
All choices of action/reaction animations play at random	Pass
Behaviors in behavior chain automatically play consecutively without overlap	Pass
All generated behavior chains play consecutively and after all are complete, Unity environment automatically stops running	Pass
Unique MakeHuman characters are spawned at random in any location in the testing area other than the location of the robotic arm	Pass
Random forward and backward reaction animations are performed at the instant a character makes contact with the robotic arm region at any angle of approach	Pass
If any type of invalid user input is detected, the Unity environment quits the run and outputs a console message outlining the issue	Pass
All user inputs are accurately read/used in the system	Pass
Robotic arm can be manually maneuvered during runtime of training environment	Pass

2. Gripper Development

2.1. Initial Gripper Issues

The gripper initially used in the robotic arm replica had several issues that became apparent at the start of a Unity run. First, while the Unity scene was playing, the gripper would detach from the robot body and fly away, only to snap back into place and repeat the process. Although no user-input was being fed into the system, the gripper acted as though it had a mind of its own. This bug caused issues in the training environment system discussed in the previous section of this paper. As discussed previously, the robotic arm region is surrounded by an invisible “Fence” with a collider that acts as a trigger. This “Fence” collider detects when it has been breached, which was meant to detect human models entering the region to trigger a reaction animation to occur. However, the gripper snapping back into place after floating away into space also seemed to trigger this “Fence” collider. As a result, even if characters were far away from the robotic arm region, if the collider was breached by the floating gripper, the characters would perform reaction animations immediately.

Figure 24: Gripper Issue #1 - Gripper Flying Away

The next issue involved the hinges of the gripper floating/spreading apart over time. This did not cause any real issues in the running of the training environment, but it was a bug that needed to be fixed to produce the most realistic robotic arm replica possible.

Figure 25: Gripper Issue #2 - Hinges Separating

2.2. Gripper Solution

Instead of scripting the movement of the gripper using the Articulation Joint system that was being used previously, it was determined that the gripper could be animated to perform grasp and detach movements. The first step was to import the gripper FBX file into Blender to begin rigging the mesh structure. This was also where the first problem was encountered. To animate the model, the mesh (3D gripper structure) itself cannot be moved in Pose Mode to form an animation clip. Instead, rigged bone structures must be moved because keyframes in animation clips record the locations, rotations, and scales of bones, not meshes. As a result, a bone structure of the model must first be made. After a bone structure is created, the mesh hinges can each be assigned to a bone so that they can accurately move together.

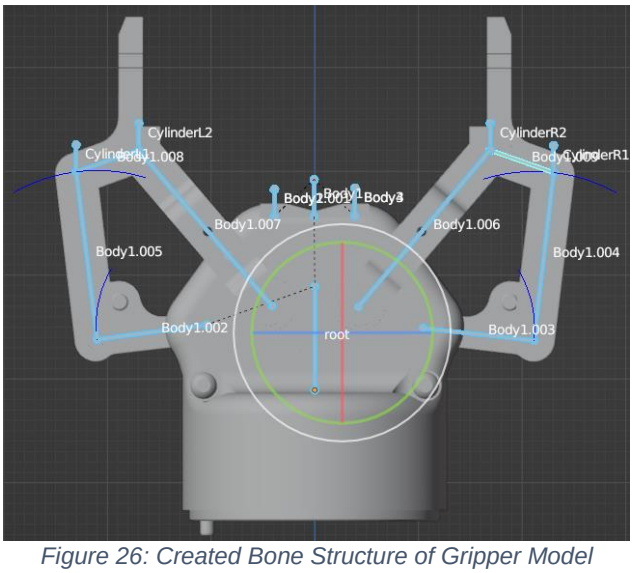


Figure 26: Created Bone Structure of Gripper Model

However, there was an issue with this process using FBX files. When importing the gripper as a FBX file, the data type of the gripper in Blender was something other than the traditional “Data Block” and as a result, assigning the mesh components to the bone structure was not an available option. The solution to this was to convert the gripper file first to an Object file (.obj) then import the gripper object file into Blender. In Blender, it was confirmed that the gripper was now of the “Data Block” type and could be set as a child of the bone structure. After assigning each mesh hinge to a bone, the gripper mesh was now entirely moveable using the rigged bone structure. For some unknown reason, after converting the gripper to an object file, a new set of cylindrical mesh components located within the hinges became visible which were never apparent when the gripper was a FBX file. These additional mesh components can be seen in Figure 27.

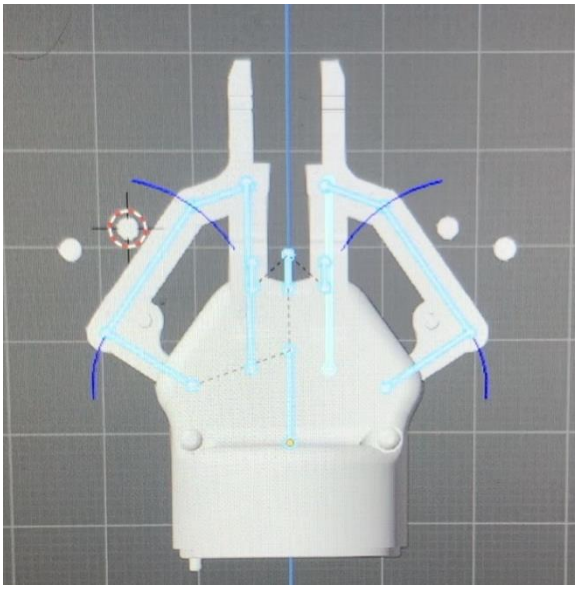


Figure 27: Additional Cylindrical Meshes in Gripper Structure

As a result, it was necessary to create new bones, assign the new mesh components to the added bones, and set the bones as children of the hinges surrounding them so that they move together and are never exposed. The additional bones are labeled “Cylinder…” in Figure 26.

After the mesh was fully rigged, it was possible to record an animation clip using keyframed locations of each bone. Each of the yellow diamonds in Figure 28 represents a keyframe that stores the location, rotation, and scale data for each bone in the rigged structure.

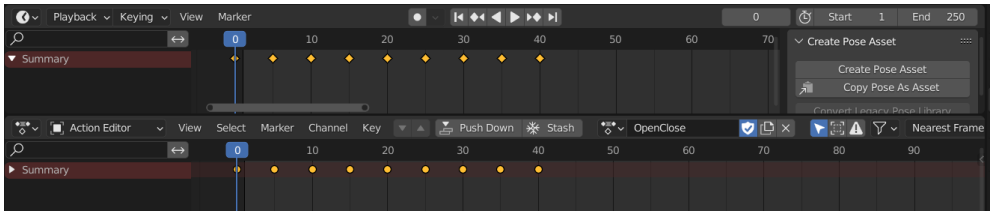


Figure 28: Animation Clip for Opening/Closing the Gripper

After the animation clip was created in Blender, the rigged gripper model along with the animation clip could be exported as a FBX file and imported back into Unity for testing. After instantiating the gripper into a Unity scene, it was possible to place an Animator component on it. An Animation Controller was created to house the open/close animation created in Blender. The Animation

Controller was then assigned to the Animator on the gripper. A script was created and placed on the gripper so that a user can control the animation status of the gripper using keys on the keyboard. After attaching the gripper to the rest of the robotic arm and adding pads to the clamps of the gripper to best resemble what the gripper looks like in real life, the Unity scene was run, and the animation was tested. Figure 29 compares the starting and final open/close positions of the gripper in both the virtual scene and real life.

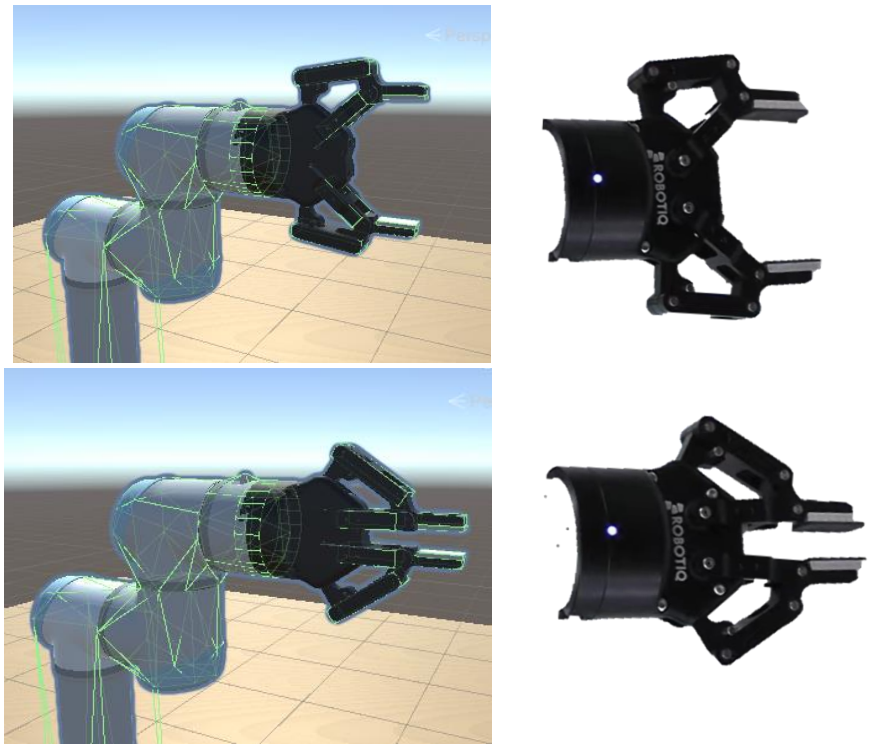


Figure 29: Open/Close Animation vs. Real Life Depiction

After verifying that the animation functioned properly, the scene was left running. After minutes of waiting, it was confirmed that the gripper no longer detached from the rest of the robot body and floated away. In addition, it was confirmed that even when repeatedly opening and closing the gripper, the hinges never separated/floated away from each other. Therefore, it was confirmed that both main gripper bugs were completely resolved. The updated robotic arm (fit with the new gripper and animation capability) was then used to replace the old robotic arm within the training environment scene.

2.3. Future Work

Although the gripper now can open and close fully without any glitches, it cannot yet physically hold objects between the clamps. To accomplish this, the gripper must be fit with colliders that act as both triggers and physical components so that the gripper can detect when an object is between the clamp fingers and as a result, pause the animation clip, leaving the gripper in a mid-animation state. However, the gripper must also contain physical colliders because for objects to react with one another in Unity, both must have colliders that will impact one another.

Currently, I have attempted to work at accomplishing this within our time in 404, but because the end of the semester is approaching, and I have only just begun exploring solutions to this

challenge, Pranav or another capstone team will likely have to take over this project segment in the future. Figure 30 displays the interaction between the colliders of the gripper clamp pads and a cube game object. Currently, the animation cannot pause when a collision is detected, but this is still a work in progress until the end of the semester.

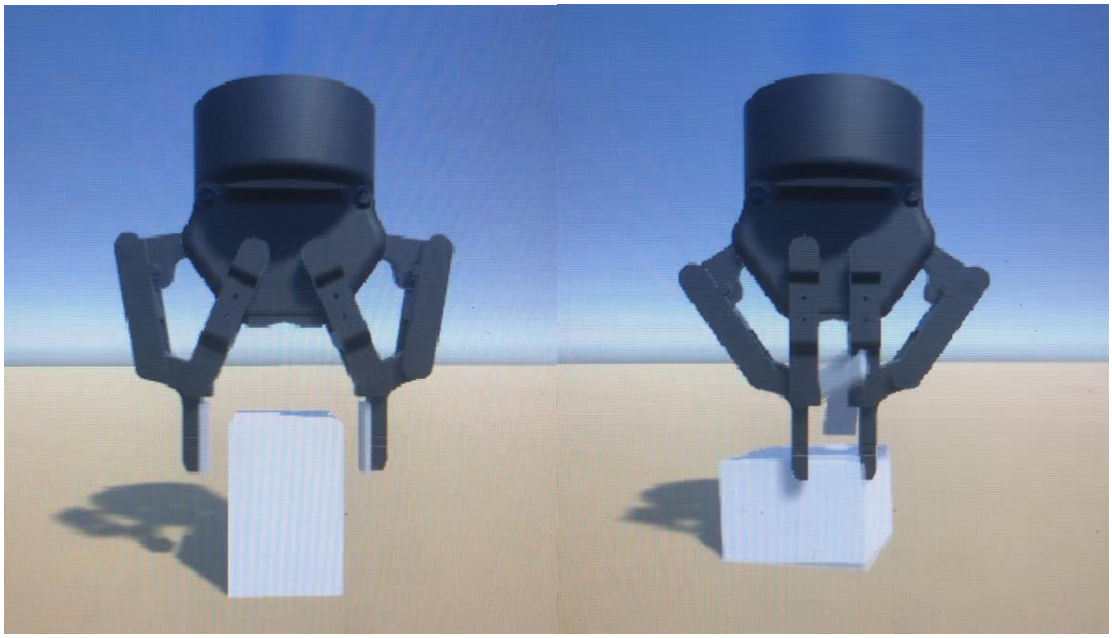


Figure 30: Current Status of Gripper Interaction with Other Objects

3. Conclusion

The robotic arm training environment has been fully tested and validated to ensure that all desired functionality has been accurately implemented. A new robotic gripper without glitches and malfunctions has been designed and directly integrated to the system. All work done in this section of our 404 project has been fully integrated with the existing progress Pranav has made in his research, proving that this environment is directly applicable and useful to Pranav in his future studies. Additionally, Pranav has reviewed and is content with the system that has been created. All code and Unity project files have been pushed to GitHub along with an extensive ReadMe file which outlines how to use the system.