The "Invisible" Keyboard

Description: A keyboard what can be controlled with 8 air-gestures for the 26 letters and 3 air-gestures for space, backspace, and enter. A machine learning model with various datasets are used to predict the word based on the 8 air-gestures. This can be helpful for people who have occupied, dirty, or difficulty using their hands steady (or make precise movements → like parkinson's disease).

The project's building blocks + what makes it unique

- **Fewer / Bigger Buttons with Prediction algorithm: HMM with a Viterbi Decoder model** (for less precision requirements)

  The Improvement over Old Tech (T9): "Unlike classic T9, which was a rigid, static lookup tree that assumed perfect physical inputs, my system uses a machine learning backend (HMM). This allows the keyboard to be *probabilistic* rather than deterministic—enabling it to dynamically handle real-time sensor noise and adapt to user movement variations."

  The Optimization Choice over Deep Learning: "While modern deep learning architectures (like Transformers or LSTMs) offer long-range contextual prediction, they require massive datasets, heavy computational footprints, and introduce high latency. For an interactive, real-time gesture keyboard, an HMM provides an ultra-lightweight, edge-compatible machine learning solution that executes in microseconds with high mathematical explainability."

  Note: Additionally, another keyboard layout is generated to provide better accuracy and prevent double-clicking with an automatic iterating keyboard layout generated where each is weighed with a score system with hill climbing algorithm.

- **Touch-free controls** (for convenience / people with disabilities)

  I was aiming for tracking a single part of someone's features, so that, firstly, it is faster/easier for the user to control one feature than multiple. Secondly, I considered making this accessible for people without all fingers / hands.

Project location: VScode -> Projects -> invisible_keyboard

Video Demonstration: https://drive.google.com/file/d/1otCho5rtW1oABHAjkKI5e8HueWalWlHN/view?usp=drive_link

Research done:

- Research Paper (to support my design decision):
  Title: EdgeWrite: A Stylus-Based Text Entry Method Designed for High Accuracy and Stability of Motion
  Authors: Jacob O. Wobbrock, Brad A. Myers, John A. Kembel (researchers in the department of computer science at CMU)

  Background: This paper is about the invention of EdgeWrite, which uses the target-based drawing technology to recognize letters instead of path-based technology like Palm OS's Graffiti.

  Key findings: The target-based letter recognition system is tested on 10 abled-bodied and 4 motor-impaired users.
  It is found that:
  For abled-bodies users, this target-based system is:

  - 18% more accurate than path-based technology
  - No impacts on speed

  For motor-impaired users (tremors, cerebral palsy), this target-system:

  - Increased accuracy rates from 31% with the path-based system to 94% with the target-based system

  Purpose for this project: This project aims to make typing in the air more accurate for abled-bodied and motor-impaired individuals. This paper showed a shift towards target-based system can be extremely helpful for accurately among both groups. So, instead of making the users target specific paths in the air, like Vulture: Mid-Air Word-Gesture Keyboard (researched by Department of Computer Science, University of Copenhagen) → extremely impressive and challenging to create for me by the way, I aimed to make a system similar to the target-based system in the air. However, since there is no actual targets in the air and no guides for users to hit that target, placing targets in specific locations would be ineffective. To replicate the stabilization benefits of target-based system without physical hardware, this system maps a box around the human's drifting finger, where the box is split into 8 sub-boxes. The program recognizes finger drifting and shifts the box with the finger. The program recognizes intentional movement and keeps the box stable:

1. **Infinite Target Bounding:** The target area effectively expands into an infinite wedge, removing the strict accuracy penalties imposed by Fitts's Law on small virtual buttons.
2. **Tremor Mitigation:** Minor micro-tremors and hand drift are mathematically filtered out, because the system only registers a keystroke when a macro-velocity threshold is crossed in a clear directional path.

This creates a highly accessible interface that accommodates the spatial uncertainty inherent to mid-air interaction.

- Research Paper (to support my design decision):
  Title: The Limits Of Expert Performance Using HierarchicMarking Menus

  Authors: Gordon Kurtenbach and William Buxton (researchers in the department of computer science at University of Toronto 1994)

  Background: The authors designed a rigorous study where users had to learn and perform commands using marking menus of varying complexities:

  - **Breadth:** They tested layouts with 4, 8, and 12 items per circle.
  - **Depth:** They tested hierarchies that went 1, 2, 3, 4, and 5 levels deep.

  Key findings:

  - Performance at 4 and 8 item layouts work well, but when it is expanded to 12 items, the accuracy dropped significantly.
  - Accuracy and fast up to the second or third depth level

  Purpose for this project: With this research papers, it is determined that there should be 8 keys to represent the 26 letters of the alphabet. As shown through this research paper, over 8-keys lead to inaccuracies, while less than 8 keys can significantly increase the difficulty of the language predicting model and increase the number of duplicates, which introduces inaccuracies. The multiple levels can be used up to two layers without great accuracy (future development for

users to introduce new words to the system because all 26 letters can be typed independent of the words around it - layer one: 8; layer two: 4)

- What is HMM? What is Viterbi algorithm? Why is this machine learning?
  Source: https://web.stanford.edu/~jurafsky/slp3/A.pdf

  What is an HMM?

  A Hidden Markov Model is a way of modeling a sequence of observations that can be seen (like words, or ice-cream counts) as being generated by a sequence of hidden states that cannot be seen directly (like part-of-speech tags, or the weather). It is built on top of a Markov chain, which assumes that predicting the next state only requires knowing the current state — everything before that is irrelevant once the current state is known.

  An HMM extends this with two key probability tables:

  - **Transition probabilities (A):** the chance of moving from one hidden state to the next (e.g., hot day → cold day)
  - **Emission/observation probabilities (B):** the chance that a given hidden state produces a particular observation (e.g., how many ice creams are eaten on a hot day vs. a cold day)

  The classic textbook framing: a climatologist can't find weather records, but does have someone's diary of how many ice creams they ate each day — the goal is to infer the hidden weather sequence from the observed ice-cream counts. Swapping "weather" for "keyboard layout group" and "ice cream count" for "which key was pressed" gives the same shape as a gesture-typing decoding problem.

  HMMs formally address three problems, and decoding is the one relevant here: given an observation sequence and an HMM, find the most probable sequence of hidden states that produced it.

  What is the Viterbi algorithm?

  Viterbi is the standard algorithm for solving that decoding

problem. In theory, every possible hidden-state sequence could be tried, its likelihood computed, and the best one selected — but the number of possible sequences grows exponentially with sequence length, making this computationally infeasible for anything real.

Instead, Viterbi is a dynamic programming algorithm: it fills in a trellis (a grid of cells, one per state per time step) left to right, where each cell represents the probability of the single most likely path that ends in that state at that time step — not the sum over all paths (that's the related Forward algorithm), but the *best* one. Each cell's value is computed by taking the maximum, over all previous states, of the previous cell's value times the transition probability times the observation probability for the current symbol.

The key detail that makes it useful for retrieving an actual sequence, not just a probability, is that Viterbi keeps a "backpointer" at every cell recording which previous state led to it. At the end, it backtraces through those pointers to reconstruct the full best-path sequence — this backtrace is how the decoded output is actually obtained, not just its likelihood.

Why is this "machine learning"?

A few reasons this counts as ML rather than just a hand-built statistical formula:

1. **The parameters are learned from data, not hand-specified.** I extracted bigram frequencies from 100,000+ text samples — that's estimating the A and B matrices from a training corpus, which is the "learning" step. (The full unsupervised version of this, for when there is no labeled training data at all, is the Forward-Backward/Baum-Welch algorithm — a special case of the Expectation-Maximization algorithm, which iteratively re-estimates the model's parameters until they converge.)
2. **It is a probabilistic model making inferences under uncertainty**, generalizing beyond exact patterns seen in training — the core statistical-learning framing, as opposed to a deterministic rule-based system.
3. **Viterbi itself is the inference/prediction step of a trained probabilistic model** — applying learned transition and emission probabilities to unseen input and producing

the model's best guess, which is exactly the train-then-predict structure that defines most ML.

- How does the Nokia T9 Layout Predictive Text work?

  T9 ("Text on 9 keys") maps the alphabet onto the digit keys 2-9 (e.g., 2 = ABC, 3 = DEF). Because each digit corresponds to 3-4 possible letters, a given key sequence is ambiguous and could match many different words. T9 resolves this with a dictionary lookup: it precomputes, for every word in its dictionary, the digit sequence that word would produce, then indexes words by that digit sequence. When a digit sequence is typed, T9 looks up every dictionary word sharing that exact sequence and presents the most frequent one as the default guess, ranked by static word-frequency counts. If the intended word isn't the top-ranked match, the user manually cycles through the other candidates sharing that digit sequence.

  T9 is a whole-word dictionary lookup ranked by static frequency. HMM + Viterbi is a sequential, context-aware probabilistic model that doesn't just search for the known word that matches this pattern.

- Research paper (for design decision implementation)

  Concept: Kinematic Feature Extraction using Angular Sector Discrimination
  Title: The Automatic Recognition of Gestures
  Author: Dean Harris Rubine (CMU thesis)

  Chapter 3: Statistical Single-Path Gesture Recognition
  Directness ratio = straight line distance (f_5) / total path length (f_8)

$$f_4 = \arctan \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$$

Distance between first and last point:

$$f_5 = \sqrt{(x_{P-1} - x_0)^2 + (y_{P-1} - y_0)^2}$$

Cosine and sine of angle between first and last point:

$$f_6 = \cos\beta = (x_{P-1} - x_0)/f_5$$
$$f_7 = \sin\beta = (y_{P-1} - y_0)/f_5$$

Total gesture length:

$$\text{Let } \Delta x_p = x_{p+1} - x_p$$
$$\Delta y_p = y_{p+1} - y_p$$

$$f_8 = \sum_{p=0}^{P-2} \sqrt{\Delta x_p^2 + \Delta y_p^2}$$

If directness ratio < 0.85, then the user is drifting. Otherwise, the movement is intentional (page 51 - 52).

Total angle traversed (derived from the dot and cross product definitions[73]):

$$\theta_p = \arctan \frac{\Delta x_p \Delta y_{p-1} - \Delta x_{p-1} \Delta y_p}{\Delta x_p \Delta x_{p-1} + \Delta y_p \Delta y_{p-1}}$$

$$f_9 = \sum_{p=1}^{P-2} \theta_p$$

$$f_{10} = \sum_{p=1}^{P-2} |\theta_p|$$

$$f_{11} = \sum_{p=1}^{P-2} \theta_p^2$$

These formulas are used to identify the arcs drawn by the user to input spaces, backspace, and enter (page 52). My program uses the angle formula to detect the angle between the segments (start to mid; mid to end). If that angle is greater than a threshold, then it is an arc.

Tools used:

- **Python** (popular programming language)
- **Mediapipe** (deep learning model trained by Google)

  I used the Hand Landmark detection model to identify the 21 key points of a hand to find the index finger.
- **OpenCV** - stands for Open Source Computer Vision Library (a widely used open-source library for real-time computer vision, image processing, and machine learning)
- **Pyautogui** - a python library that controls the keyboard and mouse
- **Numpy** - a python library for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays
- **Nltk** - Natural Language Toolkit for python programs to process human language

Datasets used:

- **Google Corpus Data** (industry standard for word, letter frequencies, bigrams - downloaded from GitHub + json that is read by the program on local files)
  - Letter frequency (for determining keyboard layout)
  - Bigrams (for determining keyboard layout)
- ***Brown University* Corpus of Present Day American English, The *webtext* Corpus, The *nps_chat* Corpus, The *twitter_samples* Corpus** (from the nltk library)
  - 100,000+ Sentences from the past decades for a balanced profile with:
    - 57,300 sentences from Brown University
      Formal grammar, complex sentence flow, foundational English transitions.
    - 10,000 sentences from webtext
      Everyday conversational text, tech-focused language (browsers, websites, laptops).
    - 10,500 chat posts from nps_chat
      Super casual slang (lol, omg, hey), natural messaging shorthand.
    - 30,000 short tweets from twitter_samples
      Ultra-modern language, highly frequent phone-centric vocabulary.

Math / Algorithms / Logic used:

- **Keyboard key grouping logic**: I decided that there are two important cases to consider when deciding the grouping of the letters in the alphabet.

  There are:
  1. **The Substitution Trap (High Severity - accurate):** If a user performs four flick gestures and the system outputs 3-5-5-2, and that code matches *both* "HOME" and "HONE" with similar probabilities, the system is fundamentally blocked. It cannot know what the user meant without interrupting them to ask for clarification. This kills typing speed and causes frustration.

  2. **The Double-Click Trap (Low Severity - high speed):** Double-clicking the same key (like typing T then H if they are on the same button) is an annoyance and slightly slows down finger cadence, but **it does not cause a decoding error**. The Hidden Markov Model can still easily figure out that button sequence 3-3 means "TH". It's a UX issue, not a system failure.

  To evaluate a keyboard letter grouping, a score is calculated where ‘the substitution trap’ is weighed x time greater than ‘the double-click trap’ because accuracy wins over faster speed.

  Calculations: The score calculations for ‘the substitution trap’ is calculated by adding up the amount duplicate words for the same user input from that the most used 10000 words in the english. The score calculations for ‘the double-click trap’ is calculated by adding up the two letter bigrams. The 10000 common words and bigrams are from *Google Corpus Data*. Both scores are presented in terms of frequencies from 0 to 1. Then they are added with the multiplier x.

  Logic: The aim is to find the keyboard grouping with the least score by iteratively improving on the keyboard layouts by finding the two sets of letters causing the most amount of duplicate inputs for different words. Then, I switched them. This process was repeated until after 5000 iterations or a peak (where the peak is defined as when the score is not decreasing by more than 0.001) is hit. With this logic, this

keyboard layout is showed to have a 22.64% improvement over the traditional keyboard layout.

- **HMM with Viterbi Decoding Algorithm**:

  HMM has two layers:
  1. **The Hidden States**: These are the true, unobserved variables you want to predict. In your project, the hidden states are the actual words the user intends to type. They are "hidden" because a single typed number could mean multiple different things.

  2. **The Observed Events**: These are the physical inputs you can actually see and record. In your project, these are the numerical button codes sent by the hand-tracker.

  The **Viterbi Decoding Algorithm** is a dynamic programming algorithm used within a **Hidden Markov Model (HMM)** to find the most likely sequence of hidden states (the *intended words*) that explains a sequence of observed events (the *typed numerical codes*).

  Instead of performing an exponential, brute-force calculation of every possible word combination in the dictionary — which would freeze the computer — Viterbi calculates probabilities step-by-step using a time-grid framework called a **trellis**.

  At each word boundary (spacebar), the algorithm evaluates two types of statistical data:

  1. **Emission Probability:** How well a word matches the current numerical code sequence.
  2. **Transition Probability:** How likely a word is to follow the *previous* word in the path (calculated using the 10,000 x 10,000 bigram matrix).

  The algorithm multiplies these values together to update a running cumulative probability tracker for each path. Viterbi applies a **survival principle**: if multiple paths lead with the same word, it only keeps the single path with the highest cumulative probability and prunes the weaker ones. This keeps the memory footprint flat and lightweight, while allowing future words to retroactively correct past ambiguities in real time.

Notable Moments:

The pivot - Day 3/4

Initial Purpose: To help people who does not have their hands clean / free (because they are painting, eating, COVID-19, etc.) to touch and type on their keyboard and move their mouse.

Initial Design: I aimed to create a computer vision tool that help me to insert input into my computer by air-drawing the letters one by one.

The problem: However, through research within the HCI (human-computer interactions) field, I discover that this method is not good. From exploring various research papers from , I realized that handwriting the letters one by one in 3D space is significantly slower (e.g. Vulture, an air-swiping to type software that uses the standard keyboard layout achieves 20 ~ 28 WPM, while handwriting struggles to achieve half of that speed), more prone to error than other input methods, and requires a lot more physical exertion and cognitive load or users already familiar with the QWERTY keyboard. Therefore, it is time to switch to another input method.

The shift: **Directional Input Model** where all letters can be written with 8 movements (up, down, left, right, up-right, up-left, down-right, down-left) where each movement stands for 3 - 4 letters. Then, computer algorithms are used to predict the words. This uses air-swiping, which is one of the fastest ways to air-input text, and reduces the amount of keys to press. The aim of this pivot is to make this typing method more accurate and faster by using less keys.

New Purpose: Initial purpose + I am removing the **requirement for steadiness**. This evolution makes the technology **faster** for expert users and **accessible** for those with motor control challenges, such as Parkinson's or age-related tremors. I am moving from 'Tracking a Finger' to 'Decoding an Intent'."

Day 8 - completion of the word/phrase prediction program

Summary: The word/phrase prediction program is done. Now all that is left is to identify the gestures. Then combine the two elements together. Surprising, there is no significant struggles to get

word/phrase prediction to work, which I initially anticipated to be the hardest part that requires the most logic and computational power.

Limitations of the Word/Phrase Prediction Model

1. Vocabulary Constraint & Static Architecture

The current model is restricted to a fixed lexicon of the top 10,000 most common English words. It operates statically, meaning it does not dynamically expand its vocabulary or adapt its probability weights based on repetitive user interactions or personal typing habits.

2. Stateless Session Processing

The system is entirely stateless across sessions. All historical HMM paths, sentence contexts, and running cumulative probabilities are held in volatile runtime memory; they are completely flushed whenever the user strikes the enter key or terminates the application execution loop.

3. Storage and Scaling Bottlenecks

While a local pipeline exists to generate and append fresh training text, continuous background dataset expansion presents physical limitations:

- **Storage Optimization:** Incremental text accumulation and n-gram matrix updates pose a scaling challenge for consumer hardware with restricted storage capacities, as an uncompressed dense $10,000 \times 10,000$ matrix scales quadratically.
- **The "Gibberish" vs. New Vocabulary Problem:** Because the primary user interface interprets physical inputs purely as a stream of numerical coordinates/digits, the program lacks an inherent semantic filter. Without a verification mechanism, the engine cannot distinguish between unintended input noise (gibberish/typos) and genuine, intentional new vocabulary words. This introduces a risk of data corruption if unverified strings are auto-saved directly into the language model datasets.

Next Steps to improve the Word/Phrase Prediction Model

- **Implement an Out-of-Vocabulary (OOV) Verification Engine:** Incorporate a lightweight, external spell-checker or dictionary API (such as PyEnchant or a local Bloom filter

layout) to act as a gatekeeper. When an unknown numerical sequence is typed, the system will verify the typed text against valid linguistic structures before authorizing it to expand code_dict.json.

- **Develop Dynamic local Adaptation:** Design a local frequency-capping update script. If a word passes the verification engine, the system will incrementally increment its weight in starting_word.json and append its local bigram occurrences into words_bigram.json using a sparse matrix format to prevent storage bloat.

Day 10 - completion of everything

Summary: The end-to-end integration of the touchless text-entry system is complete. Developing a robust spatial gesture-recognition pipeline presented significantly greater engineering challenges than implementing the predictive language model. While predictive text relies on structured, discrete inputs, real-time computer vision must parse human hand gestures that exhibit massive spatial variance, velocity fluctuations, and behavioral freedom.

Limitations of the Product

1. Gesture Recognition Inaccuracy & Signal Cross-Talk

The primary technical limitation lies in spatial segmentation cross-talk. Fast, high-velocity straight strokes (intended for numeric selection) are occasionally misclassified as curved trajectories (intended for [SPACE] or [BACKSPACE]), or vice-versa. Additionally, tracking subtle depth translations along the Z-axis for the [ENTER] command remains highly sensitive, occasionally leading to accidental triggers during micro-movements. However, once a numeric sequence is successfully captured, the Viterbi decoding and bigram language model layers perform exceptionally well, consistently mapping the digit streams to the intended words.

2. Absence of Multi-Modal Haptic and Tactile Feedback

The current iteration relies entirely on visual feedback, requiring the user to continuously monitor the screen overlay to view the active character sequence, top predictions, and sentence state. In traditional typing, physical keyboard switches provide tactile and auditory confirmation. The lack of non-visual feedback in mid-air typing increases the user's cognitive load;

incorporating auditory clicks or localized haptic feedback represents a critical area for future user-experience research.

3. Throughput Constraints and Cooldown Latency

To mitigate signal noise and isolate consecutive strokes from residual hand momentum, the system enforces a strict 1-second non-blocking cooldown lockout between gestures. While empirical testing proved that a 1-second window is the optimal threshold for filtering out duplicate frames and stabilizing accuracy, it introduces a hard cap on input throughput (words per minute). Optimizing this stabilization window is essential for achieving fluid, high-speed text entry.

Next Steps for the Product

- **Dynamic Gesture Filtering:** Replace fixed pixel thresholds with adaptive filtering (such as a Kalman filter) to smoothly isolate intentional trajectory changes from high-frequency webcam sensor noise.
- **Machine Learning Micro-Classifiers:** Train a lightweight, local neural network (e.g., an SVM or simple MLP) on the 13 Rubine features of your custom strokes to classify lines, arcs, and sharp corners, replacing the hardcoded geometric thresholds.
- **Multi-Modal UX Integration:** Integrate auditory cues (spatialized tone clicks) or explore hardware integrations for haptic feedback to minimize user eye strain and simulate physical button actuation.

Timeline

Day 1: Spent a day installing Mediapipe because the current Mediapipe version is not updated to match the Python 3.14.4 environment yet.

Day 2: Developed the hand tracking tool using 21 landmarks on the hand on hand_tracker.py.

Day 3 (the pivot): Researched into natural hand movements to differentiate between active strokes, transition phrases, and resting/hovering to make an interface that is more intuitive for users (aka similar to normal writing motions). I discover that “From exploring various research papers, I realized that handwriting the letters one by one in 3D space is significantly slower, more prone to error than other input methods, and requires a lot more physical exertion and cognitive load or users already familiar with the QWERTY keyboard. Therefore, it is time to switch to another input method.”

Day 4 (the pivot): Decided on the Directional Input Model where all letters can be written with 8 movements (up, down, left, right, up-right, up-left, down-right, down-left) where each movement stands for 3 - 4 letters. Then, computer algorithms are used to predict the words.

+ installed / explored pyautogui with speed detection of fingers on hand_drawing.py

Trouble: I tried to identify when the user is swiping across the screen and wanted to use that motion as an indication for the computer to clear the screen. I realized that even a small tremor can be within the ‘speed of swiping’ range that I was aiming for.

Solution: Add a distance requirement on top of the current speed requirement, so that tremors do not trigger the output.

Day 5 (keyboard-layout determination): A program is ran for determining a better keyboard layout as described in *Keyboard key grouping logic* under *Math / Algorithms / Logic used.* This keyboard layout was improved with the baseline being the standard layout from the traditional Nokia phone T9 layout with the purpose of decreasing overlapping words with the same numerical code and double clicking with a hill climbing technique.

Day 6 (direct keyboard translation): With the keyboard layout set, a dictionary is used to sort the top 10,000 words in English into what their numerical code is. With that, every numerical code of a word /

phrase / sentence written can be translated into a list of text options that matches the numerical code.

Day 7 (create the datasets for machine learning algorithm): For the program to know what word to start with and what words comes after a previous word from the direct keyboard translation list, I needed two datasets ‘starting_word.json’ and ‘words_bigram.json’.

‘starting_word.json’ is a dictionary of in the format of {*word*: *freq*} where the freq represents the likelihood of *word* being the starting word from 0 to 1.

‘words_bigram.json’ is a matrix of approximately 10,000 x 10,000 where bigram[*prev_word*][*cur_word*] gives the likelihood of *cur_word* coming after *prev_word*.

The two datasets are created by analyzing 100,000+ sentences from *Brown University* Corpus of Present Day American English, The *webtext* Corpus, The *nps_chat* Corpus, The *twitter_samples* Corpus (from the nltk library), which gives a range of formal to informal / daily language.

Note: these datasets are auto-generated by the program, can can be easily updated with new data.

Day 8 (HMM with a Viterbi Decoder model implementation): Using the probability datasets from *Day 7: a Hidden Markov Model (HMM)* framework was built to resolve structural typing ambiguities. Rather than processing text using a greedy word-by-word matching sequence, a localized Viterbi tracking algorithm was developed to hold a rolling matrix trellis of survival paths in runtime memory. When numerical inputs are passed, the decoder calculates cumulative path trajectories by multiplying emission probabilities (layout code lookup matches) with transition values ($P(\text{word}_t \mid \text{word}_{t-1})$ derived from the bigram matrix). To keep runtime execution lightweight, weak paths are continually pruned, retaining only the top alternative sentence sequences. Furthermore, this dynamic trellis framework was successfully adapted to handle prefix-based autocomplete word predictions during live character accumulation, alongside an inverse division rollback function to handle cross-word boundary backspaces. This design ensures that text display outputs can seamlessly adapt and rewrite backward in real-time as subsequent contextual clues are keyed in.

Day 9 (identifying hand gestures): Reading the ‘holy grill’ of the hand gesture recognition paper from CMU (link), I wrote formulas to differentiate between intentional and non-intentional movements, and

arc and straight lines to identify the 8 numbers (for the 26 letters) and clockwise/counter-clockwise (for ‘space’/’backspace’) movements. Additionally, I analyze the the z-axis (the depth from the camera) relative to the wrist to analyze the ‘enter’ movement.

Day 10 (combining everything): Combining everything was harder than expected. I originally thought that combining the functions I wrote would not require any alterations of the functions at all. However, I reached a point in the combining process that I was left with two functions with while loops inside that is intended to drive the whole program, where I had to decide to alter one function to fit it within the other. Then, I made the program a bit more presentable with information showing on the screen for better user experience.