A Gesture based programming scheme for the Crazyflie Micro-UAV

Owen Levin[1] and Dennis Melamed[2]

***Abstract*— In this work a gesture language for basic UAV programming is developed, and an implementation written using a Leap Motion Controller as an input device. The language is interpreted and commands sent to the UAV through the ROS framework, allowing different UAVs to be used with the same pipeline.**

I. Introduction

Programming a robot is generally a task undertaken using a keyboard and screen, sitting at a desk. However, as robots become more and more prevalent in all fields of human life, those developing them venture into the field with increasing frequency. The field can range from standard office buildings, where access to a desk with a computer to reprogram a robot is very easy, to fields of mud with nowhere to set a keyboard, to underwater where even getting a keyboard working is a huge challenge. Thus, a different way of programming a robot that did not require a stable surface to set a keyboard may be useful in difficult field conditions. Gesture based programming is one such method.

To many users of computer systems, gesture control conjures thoughts of "Minority Report" style hand waving and immediate reactions intimately connected to body movements. While this is enjoyable in films, the practicality of such a system is questionable in robotics applications. However, due to advances in tracking technology, extremely accurate small scale recognition of parts of the body such as hands is within the realm of possibility. Developing a gesture based programming scheme for such a device would simplify the requirements for field programming to a processor, a small screen, and the gesture recognition device itself, all easily placed in a small unit that can be carried with the user and used from a standing position.

This is by no means the first visual programming scheme. Previously, tracked movements patterns have been used to program as in [6], and simple glyphs that can be easily detected and tracked as in [1] have been combined to create complex programming schemes. Our work differs in how we collect gesture data, classify it, that we only use 22 static hand gestures, and that we make no attempt to implement complex programming concepts beyond loops and defining macros. In other applications, human gestures have been used in attempt to demonstrate what a (typically industrial) robot should do to accomplish a task as in [4]. In such an application, machine learning methods are employed to teach the robot how to mimic the effect of a human task/gesture once it is recognized.

The work undertaken here seeks to develop a gesture-based control scheme for the Crazyflie micro-UAV. Our framework provides a simple way to explicitly program path plans using sequences of movements in the x, y, and z directions. Gesture input is provided by the Leap Motion Controller, an innovative combination of sensors which allows super-precise hand tracking. The control scheme forms a rudimentary programming language, allowing users to write, compile, and run macros made up of combinations of simple UAV motions. Other uses of the Leap have been used by e.g. Tsarouchi et al. in tandem with a Kinect to map body and hand gestures to movements of a robotic arm with a gripper [5], though unlike our work, Tsarouchi et al. did not allow for defining and calling macros or variables.

The rest of this paper is organized as follows. First we describe the gesture programming language we developed. Next is a discussion of the hardware used, followed by a walkthrough of the software stack for the language. We conclude by presenting our results; possible avenues for future work are also proposed.

II. Gesture Language

The gesture language is comprised of 22 gestures shown (labelled images of these are shown in the Appendix). The framework for our gesture language is built around defining macros, which may be called to build more complex macros, or run directly by the micro UAV. The syntax allows for three options whenever no other commands are being processed. We

[1]Owen Levin is with the School of Mathematics and [2]Dennis Melamed is with the Department of Electrical and Computer Engineering, University of Minnesota, MN 55455, USA

call such a situation the "Idle environment," and from it one may either define a numerical variable, start recording a new macro, or run a previously recorded macro.

A. Setting Numerical Variables

To define a numerical variable, the user would perform the `SetNumberVar` gesture. Afterward, the user may perform any arbitrary sequence of gestures not containing the `END` gesture. This sequence of gestures is the name of the variable being set. To finish naming the variable, the user performs the `END` gesture. Now there are two options; first, the user may simply perform `END` again, in which case the value would be unset, so the variable is deleted and nothing happens. Otherwise, the user may enter a number which will be the variable's value.

B. Numbers

Numbers are ubiquitous in our gesture language as values of variables, names of macros, and parameters for actions. All numbers we use are integers, and each one is defined digit by digit. That is, to start a number the user inputs a `Digit` gesture followed by any digit in $\{$`0,1,2,3,4,5,6,7,8,9`$\}$. Then, the user may repeat the digit number sequence arbitrarily many times to define numbers of arbitrary length, completed by an `END` gesture. In addition, the `Negate` gesture can be performed at any time after the first `Digit` gesture to swap the number's sign. Example: The number 135 could be constructed many ways, such as by entering
`[Digit, 1, Digit, 3, Digit, 5]` or also by `[Digit, Negate, 1, Digit, Negate, 3, Digit, 5]`, but not by
`[Negate, Digit, Negate, 1, Digit, 3, Digit, 5]` because a number must begin with a `Digit` call. To be sure, any even number of `Negates` (including 0) will result in a positive number, and any odd number of `Negate` will result in a negative number.

Beyond setting numbers directly, if a numerical variable has previously been defined one may enter it anywhere a number would normally be entered. This is done by using the `CallNumberVar` gesture in place of the initial `Digit` gesture. Then, the user enters the name of the number they wish to use followed by an `END` gesture to complete the variable call. In the current setup for the language, `CallNumberVar` is overloaded with the same gesture as `Negate`, which is why (along with ease of implementation) `Negate` cannot be used as the first gesture in defining a number directly.

C. Recording Macros

To record a macro, the user performs the `RecordMacro` gesture, followed by a number to save the macro to, followed by a sequence of actions. These actions may be waiting (`Wait`), moving in the x, y, or z directions (`MoveX, MoveY, MoveZ`), or calling a previously recorded macro (`CallMacro`). Each of these actions takes a numerical parameter. For waiting, the parameter is the number of seconds to wait in place. Movements' parameters are the number of decimeters to move in the desired direction, and the number for a call macro refers to which macro should be called. The units for movements and wait time could be altered in the run-time node of our implementation if desired.

Beyond just listing sequences of actions, the language allows for loops by performing `Repeat` followed by a number (the number of times to repeat), then a sequence of actions or macro calls that should be looped. The `END` gesture closes to scope of a loop or recording of a macro.

D. Running Recorded Macros

Finally, the `Run` gesture allows one to run any macro that has been previously recorded and, like calling a macro, takes a single numerical parameter corresponding to which macro should be run. A general schematic of the command flow for the language is provided in Figure 1

III. Hardware

The micro-UAV we desired to program using our the gestural language is the Crazyflie micro-UAV. The Crazyflie is 27g, has a 15g payload, and has a downward facing camera for localization. To capture the gestural inputs, we used a Leap Motion Controller. The Leap uses three LEDs to output 850 nanometer light to the nearby area. To capture data, it uses stereo cameras to take two grayscale images thresholding for the wavelength of light its LEDs produce. These images are used to localize points in 3D space near the Leap, in particular 29 key points of the user's hand.

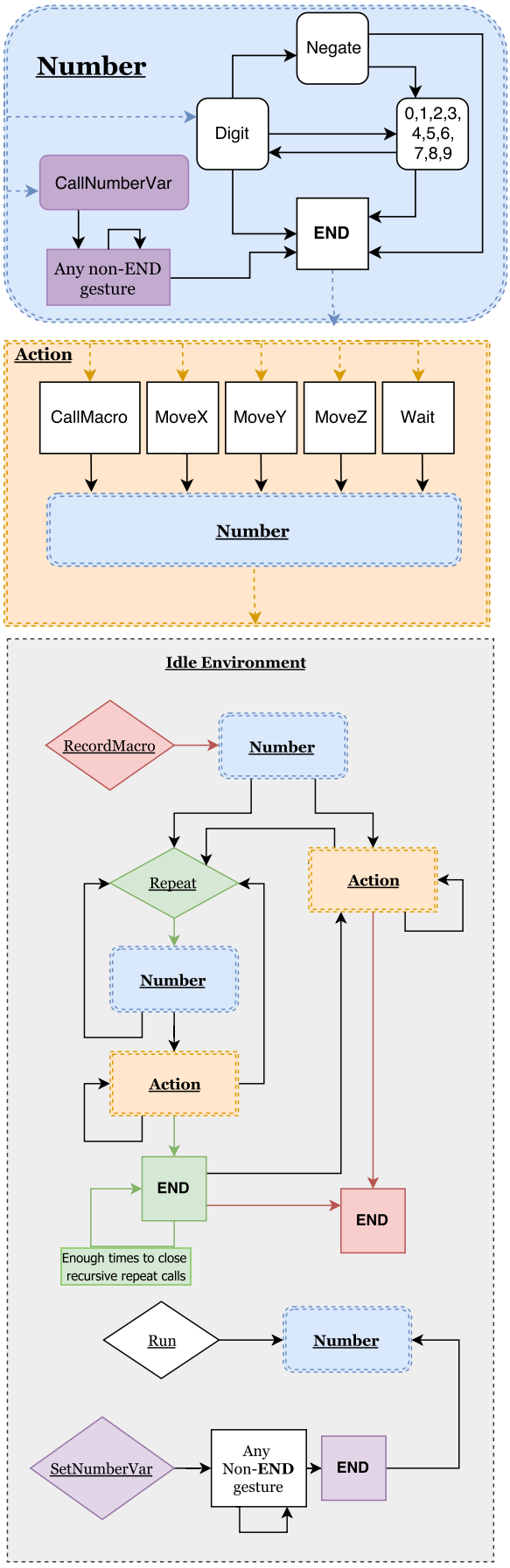


Fig. 1. Top: command flow for numerical parameters, Middle: command flow for an action, Bottom: command flow overall

IV. Gesture Recognition

The Leap Motion provides a ROS node which publishes the (x, y, z) coordinates of 29 points on the user's hand and arm. To utilize this information, we trained a 200-Nearest-Neighbor classifier to recognize which pointsets corresponded to which of the 22 gestures comprising the gesture language. This classifier appeared most effective of the several different parameters and types of classifiers we attempted to use, though it is still imperfect. Using this classifier, we designed a gesture recognizing ROS node, which subscribes to and classifies the Leap's data. Before publishing which gesture is being recognized, the previous 100 gestures are looked at and the mode of the classified gestures is identified. The currently recognized gesture is published when the mode is consistent for a few seconds. That way, mis-classifications are published less frequently, and the ease of programming is improved.

V. Compilation

The next node in the pipeline is the Compiler Node. The compiler subscribes to the current gesture output by the gesture recognizer and processes it according to the language syntax. The implementation is as follows.

First, the compiler keeps track of stack of scopes as `ScopeBlocks` with a base Idle scope at the bottom of the stack. Each `ScopeBlock` has parameters for the type of scope (recording or repeating) as well as a numerical parameter, and a list of `ActionBlocks` corresponding to which actions are contained in that scope. Each `ActionBlock` has parameters for the type of action it corresponds to and its numerical parameters. In addition to the scope stack, the compiler has Boolean flags for whether variables are being called or recorded.

To test whether a gesture is allowed, the compiler consults one of several appropriate dictionaries depending on the current scope and which flags are set. Each of these dictionaries maps a previous gesture to legal commands that may be used afterward. If a gesture is deemed illegal, it is ignored. Otherwise it is processed by the compiler according to the language syntax.

A. Loops and Macro Calling

In processing, whenever a loop containing a sequence of actions is `END`ed, the loop is unrolled, and each action in it is added appropriately to the list of

actions `ScopeBlock` in which the repeat was called. Similarly, as the loop is unrolled, any macro being called is unrolled to add whichever actions the the called macro is made up of to the `ScopeBlock`.

B. Storing Numerical Variables

Numerical variables which the user defines are added to a python-pickled file containing a dictionary mapping tuples of gestures to numbers. This allows variables to persist between programming sessions.

C. Writing Recorded Macros

When a macro recording `ScopeBlock` is `END`ed, the list of actions is written to a `Macro#.csv` file. This file is human readable, and a user can easily write any macros they may think will be useful to call or run in advance. Suppose we wanted to program the UAV to move in the following fashion.

```
For(index = 0; index < 3; index ++){
  Move 10 decimeters in the z direction;
  Wait 3 seconds;
  For (index2 = 0; index2 < 2; index2 ++){
    Move 1 decimeter in the x direction;
    Move 1 decimeter in the y direction;
  }
}
```

Then the corresponding macro file would have the following contents.

```
z 10
w 3
x 1
y 1
x 1
y 1
z 10
w 3
x 1
y 1
x 1
y 1
z 10
w 3
x 1
y 1
x 1
y 1
```

D. Compiler Output

Lastly, the compiler constantly publishes two messages. The first tells the number of macro runs that have been sent to the run-time so no macro is accidentally run twice, as well as which macro was most recently run. The second message contains most of the data the compiler keeps track of so that the GUI may utilize the information to aid programming

E. Other Compilation Notes

If the previous gesture was a 0-9, one can perform another 0-9 gesture to overwrite the current digit as the new 0-9 gesture as long as the `Digit` or `END` gestures have not been entered. Similarly, action gestures followed by other action gestures will have the action type overwritten prior to the `Digit` command. Whenever an action (which is not calling a macro) or a repeat is used with parameter 0, the action block or repeat scope is deleted since repeating 0 times, moving 0 distance and waiting 0 time are effectively doing nothing.

VI. Runtime

Following compilation, the recorded macros must be interpreted and transmitted to the UAV at the appropriate time. The runtime node acts in much the same way as, for example, the Java Runtime does. The ”byte code” in the form of the macro comma separated variable file is loaded into the runtime upon a request from the compiler. This request, in the form of a change on a topic published by the compiler and monitored by the runtime, occurs when the use executes a

```
[Run, Digit,
(desired macro number), END]
```

sequence. Upon being loaded into the runtime, the macro file is broken down into individual commands, and those commands are added to the runtime queue. This operation occurs in parallel with the processing of the front of the queue. In the processing thread, a command is taken off the front of the queue upon successful completion of the previous command. Prior to any macros running the previous “command” is set to already-completed. The new current command is interpreted as either an x,y,z motion or a wait period. The appropriate modification in the goal location for the UAV is made based on this command, or processing is halted for the specified period if the command is to wait.

The runtime is the only portion of the pipeline that is in a feedback loop with the UAV, allowing for maximum modularity when changing from one UAV to another. In order to determine when to run the next command, the transform from the UAV to to the world frame is constantly polled, and when it is within positional error (specified at compile time of the runtime node) of the goal position, the next command is pulled off the queue and interpreted. When no commands are left in the queue, the UAV maintains its position, allowing for data to be collected or other UAV-based operations.

Similarly to a more complex programming language, the runtime is the final stage of the gesture language pipeline. The input the UAV receives is in the form of a goal position in the world frame. The interpretation of this information is up to the UAV, however the frames the UAV considers its own and its world's must be consistent with those monitored in the runtime for accurate motion. The software stack beyond this point is discussed in later sections of this paper.

VII. Graphical User Interface

The graphical user interface, or GUI, for the pipeline is not a necessary part of the system, and the UAV can be commanded with pure gesture input and feedback based on the UAV's motion. However, this is very difficult due to the likely-hood of misinterpreted gestures and the requirement to memorize the gesture syntax in its entirety. The GUI provides an easy to use interface for monitoring all parts of the software stack. It runs as a separate node, allowing it to be launched and closed as necessary while keeping other parts of the stack operational.

The leftmost part of the GUI window provides a list of macros currently stored in the

```
crazy_frog/macros/
```

directory. Clicking on a macro will display its contents in the window directly to the directory display window's right.

Along the top of the display are three panels, displaying information about compilation. The first shows the name of the current scope, and the window to its right shows the contents of the current scope. Possible scopes, as mentioned previously, include Idle (prior to any recording), the Record Macro scope (which includes the macro number being recorded once that is specified), and the Repeat Scope (which contains the commands to be looped). Upon exiting a repeat scope, the containing Record or Repeat scope will contain the unrolled loop of the commands in the exited scope. The final panel displays the commmand (x,y,z or wait) currently being built. It also displays the number parameter being attached to it. The number portion of this panel will display any number being built, a useful feature when, for example, selecting a macro to run.

The middle portion of the display also contains three panels, focused on gesture recognition. The first panel displays a list of previously recognized commands. This allows a user to quickly remember a previously input sequence, or look at the gestures the compiler has received. The middle panel shows legal next gestures based on the previous gesture, removing the requirement of the user memorizing the entirety of the language syntax, similar to an Integrated Development Environment's "suggestion" capability. The final panel in the middle row displays the human-readable gesture currently being recognized by the gesture recognition part of the stack, allowing misclassifications to be corrected immediately.

The bottom row of panels displays information relevant to the runtime, as well as information about the UAV. The runtime queue is continuously updated and displayed in the leftmost panel, allowing a user to see the next motions the UAV should be performing. The next panel shows the current command being run, allowing debugging of the UAV if it is not flying correctly. Finally, the current world-to-UAV transform is displayed alongside the goal world-to-UAV transform in the final panel.

VIII. Results

A. Simulation

In order to test the gesture pipeline, a simple V-REP simulation was developed. This simulation models a quadcopter and the physics surrounding it. The simulation broadcasts the transform between the quadcopter and the world frame within the simulation, providing the feedback the runtime requires. The simulation implements a simple quadcopter controller, moving towards a green sphere at all times. This green sphere has been set up to monitor the goal position broadcast by the runtime and instantaneously move there. This

allows visualization of the recorded macro as the simulation runs.

Provided a consistent hand height is maintained and the user is knowledgeable in how to perform each gesture, relatively consistent recognition of all gestures was achieved. Macros were able to be recorded and run successfully by a user who had the gesture classifier trained to their hand, as well as by users with similar hands.

Consistency of recognition continues to be a problem for this pipeline. The K Nearest Neighbors classifier, while simple, is not very robust to variations in how a gesture is performed. Correct height above the sensor, precise positioning of fingers, and proper hand angle relative to the sensor are all needed for a correct recognition of some gestures.

An additional problem is mis-recognition of gestures. For many gestures this is not an issue, and for some the structure of the gesture syntax causes mis-recognition to be a non-issue. However, some gestures are consistently mis-classified as another, for example END and 6. Thus, if a user is attempting to end a number, the last digit could be inadvertently changed to 6, causing problems.

B. *Crazyflie*

We did not succeed with implementation on the UAV due to difficulties in calibrating the UAV system and achieving stable flight. Since the software stack for the gesture based programming scheme ends with providing a goal position for the UAV to move to, the UAV controller to achieve this goal is unique to each robot. A stable transform to the UAV in a world frame with its origin at the initial pose of the UAV can be determined using the downwards facing camera mounted on the robot. The transform from the world frame to the camera frame was achieved using the Robotics and Perception Group's svo_ros node [2]. In order to determine the transform from the camera frame to the UAV's IMU, which the UAV controller treats as its center, a full calibration must be run on the two devices. Getting an accurate calibration has proven difficult, as the tools to achieve this are not very well documented. Following this, a lengthy period of trim and weight adjustment is necessary to achieve stable UAV flight, particularly with a robot of this small size. The motion and motor controllers for the UAV are implemented in the control software [3], thus the key impediment to using the UAV is the camera-IMU calibration. Due to the modularity of this project, if stable flight and accurate motion of the UAV can be achieved, then the already tested gesture processing pipeline will be able to be easily integrated.

IX. Future Work

The primary focus of future work on this project will be in implementing the controller on the physical micro-UAV. Once the camera-IMU calibration is complete, tests can be performed on how well the gesture recognition pipeline performs in real life situations, such as in the field or even for rapid prototyping of simple robot motion schemes.

While powerful, the extensibility of the compiler could be improved by redesigning it in a more scope oriented manner, allowing users more fine grained control over how specifically they structure their programs.

Implementation of all nodes in C++ would be a considerable time investment, but would likely yield dramatic returns in terms of processing speed at every step of the pipeline. This would help particularly in the case of gesture recognition, where a more responsive recognition node would allow faster programming by the user.

From a language point of view, developing further feedback loops with the UAV while maintaining UAV agnosticism would be a fruitful area for further work. If combined with the implementation of an if/else clause in the gesture syntax, the use of sensor information from the UAV would allow programming UAVs for more autonomous flight. Automatic landing when the battery is low, collision avoidance, and simple flight path decision making are possibilities once sensor data is incorporated into program flow.

Another possibility is converting `[MoveX, MoveY, MoveZ]` into a goal position consisting of (Latitude, Longitude, height). For a UAV that already has SLAM or other localization and sensing methods, this would be a simple modification for the compiler and run-time, but potentially provide more versatility.

References

[1] G. Dudek, J. Sattar, and A. Xu. A visual language for robot control and programming: A human-interface study. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2507–2513, April 2007.

[2] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. SVO: Fast semi-direct monocular visual odometry. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[3] Wolfgang Hoenig, Christina Milanes, Lisa Scaria, Thai Phan, Mark Bolas, and Nora Ayanian. Mixed reality for robotics. In *IEEE/RSJ Intl Conf. Intelligent Robots and Systems*, pages 5382 – 5387, Hamburg, Germany, Sept 2015.

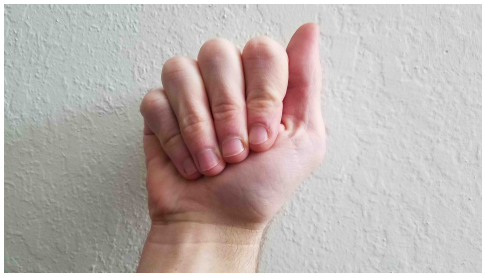
[4] Hsien-I Lin and Y. P. Chiang. Understanding human hand gestures for learning robot pick-and-place tasks. *International Journal of Advanced Robotic Systems*, 12(5):49, 2015.

[5] Panagiota Tsarouchi, Athanasios Athanasatos, Sotiris Makris, Xenofon Chatzigeorgiou, and George Chryssolouris. High level robot programming using body and hand gestures. *Procedia CIRP*, 55(Supplement C):1 – 5, 2016. 5th CIRP Global Web Conference - Research and Innovation for Future Production (CIRPe 2016).

[6] A. Xu, G. Dudek, and J. Sattar. A natural gesture interface for operating robotic systems. In *2008 IEEE International Conference on Robotics and Automation*, pages 3557–3563, May 2008.

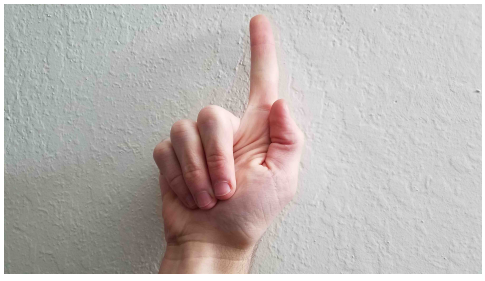
X. Appendix

Gestures in the images are from the point of view of the Leap Motion sensor, which usually rests on a table between 6 and 7 inches below the hand.

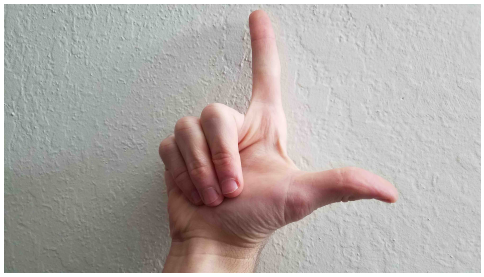


0

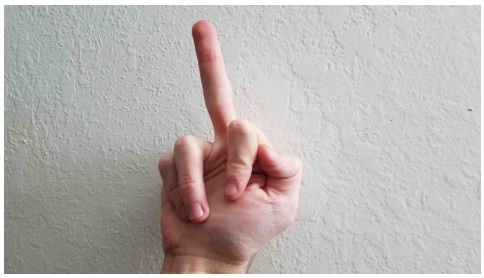
1



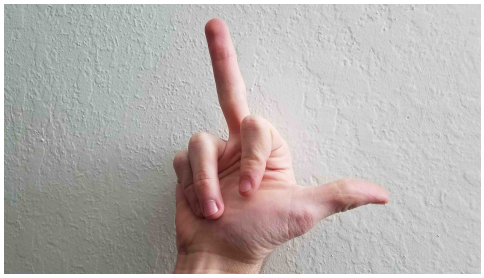
2



3

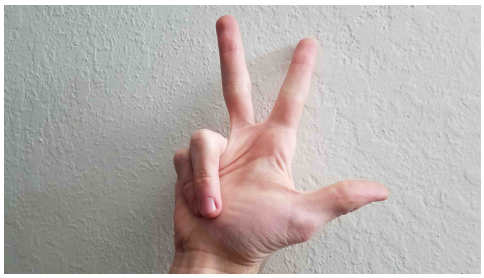


4

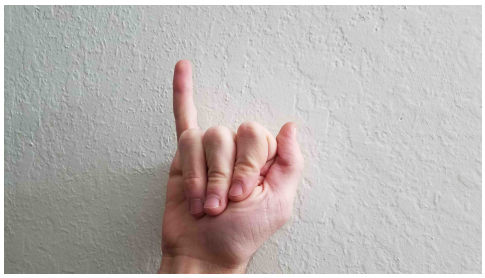


5

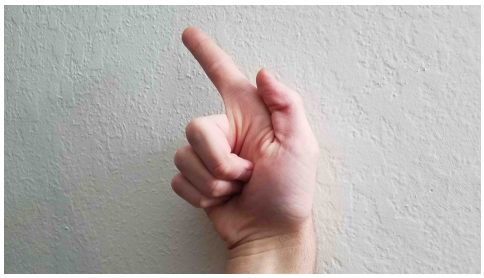
6



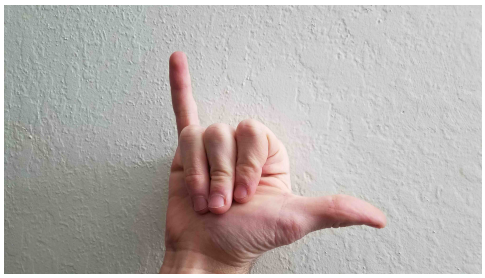
7



8

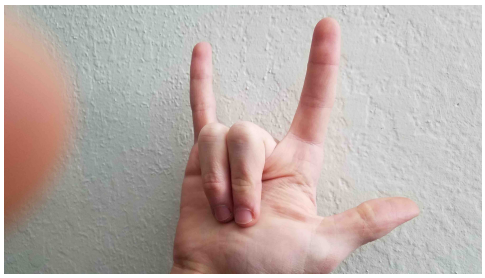


MoveX



9

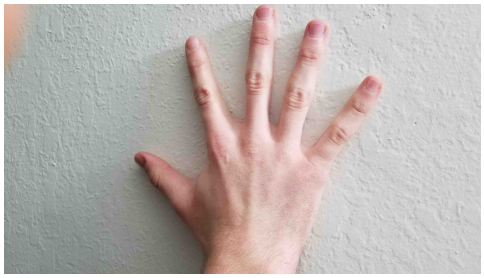
MoveY



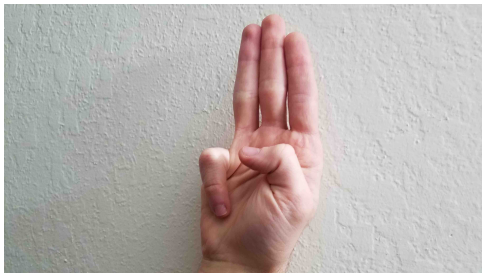
CallMacro

MoveZ

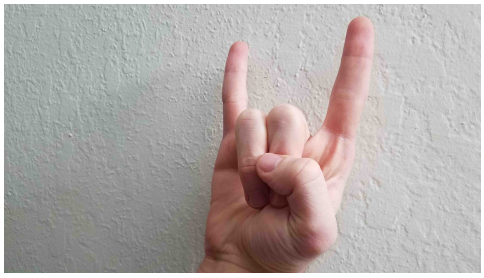
Digit



Negate / CallNumberVar

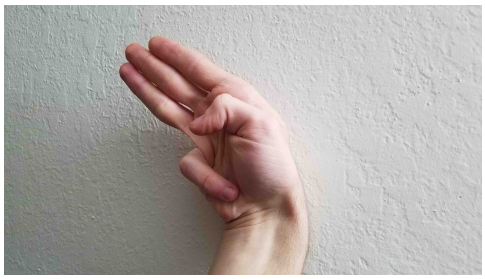


END

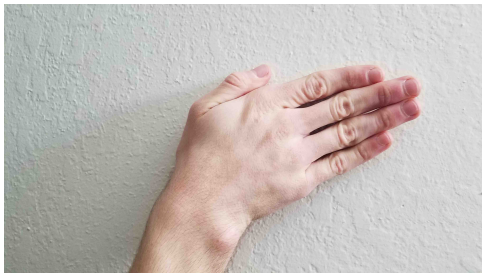


RecordMacro

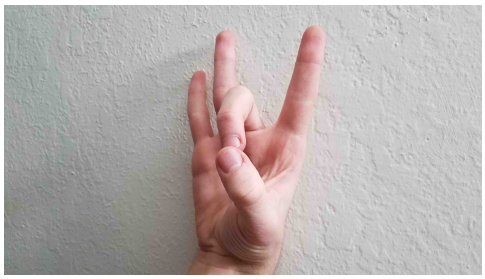
Repeat



Run



SetNumberVar



Wait