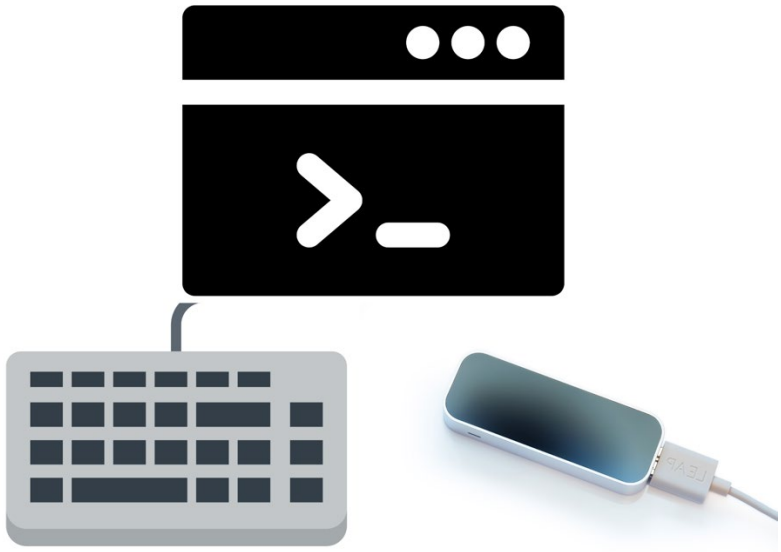
GesTerm

Long Phi Nguyen

6.835 – Intelligent Multimodal User Interfaces
Spring 2019



What if you could swipe or draw an S or tap your finger to run a command in your terminal?

I built a general-purpose gestural macro tool, GesTerm, to enhance the terminal user experience. I combined Python 3.7 with the Leap Motion sensor's ability to track users' hands. Users specify what macros they want to use and the terminal commands they want associated with these macros. By detecting these macros through the Leap Motion SDK, GesTerm triggers the relevant commands seamlessly and alerts the user. GesTerm currently supports circle and swipe gestures.

GesTerm's seamless operation works well in tandem with native keyboard input, and gestures were detected extremely well, provided we have a controlled environment with little visual noise.

An area of exploration in the future is machine learning for *custom gestures* that users can specify and train the system with. This would provide a more intuitive, natural experience.

GesTerm

Introduction and Overview

The computer terminal marked the beginning of the graphical user interface's dominance in modern computing. More than five decades after its inception, the terminal is still a preferred environment for many software developers who want the simplicity of a text-based UI.

With modern software growing more and more complex, terminal-based programs often resort to cryptic keyboard shortcuts for functionality that cannot be addressed in other ways. Integrated solutions such as trackpad/mouse-based scrolling in terminal-based text editors like vim exist, but these do not address potential multimodal interactions involving gestures or speech. Other solutions include Apple's Touch Bar, the functionality of which depends on the specific terminal application used.

Overall, the current keyboard shortcut scheme employed by many terminal-based applications is inefficient and far too complex for the average user. An interesting goal would be to produce an application that allows the user to store gestural macros for use at any time when working on the command line. The task, then: whenever a user wants to avoid the overhead of the touch modality (working on the keyboard), he can perform the macro associated with whatever command he would like to run.

This task is interesting, as no previous work has been done to support gesture-based macros on the command line. I wanted to see if productivity could indeed be boosted through gestures, especially with our heavy focus this term in class on gesture detection.

System overview

What does it do?

GesTerm triggers terminal commands based on gestures performed by the user. It does so by generating keypresses on the operating system through Python. These gestures are first specified by the user for the daemon to run:

```
python3 gesterm.py swipe 'cmd1' circle 'cmd2'
```

is the format for running GesTerm as a daemon, where `cmd1` and `cmd2` are associated with the two supported gestures. To inform the user that a gesture is in progress or has ended, GesTerm sends native macOS alerts (no support for Windows 10 at this time) that, in particular, give the gesture in action and, if it is a swipe gesture, the *xyz* direction the gesture was performed in. Figure 1 shows various alerts that GesTerm triggers.

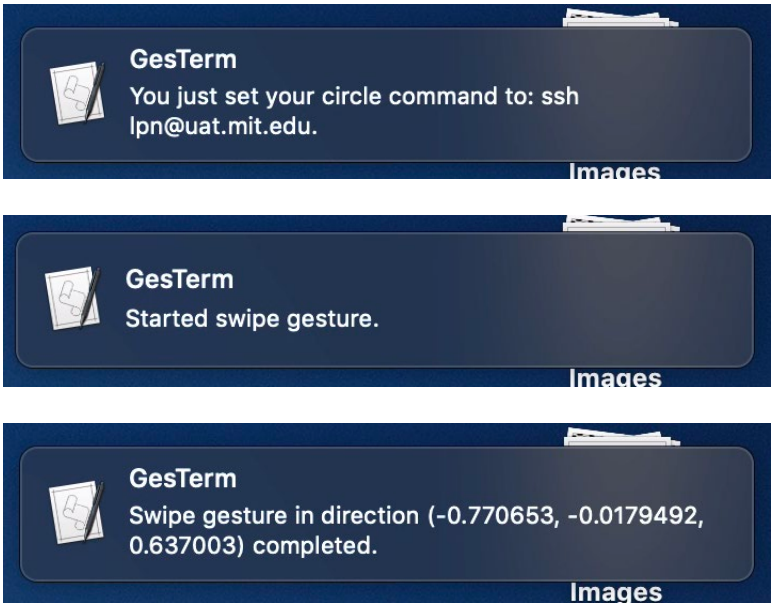


Figure 1: Various alerts shown by GesTerm when macros are set and gestures are performed.

When the user swipes or draws a circle, the commands they specified at initialization are executed in their terminal application.

Suppose a user wanted to persist an `ssh` command through GesTerm to avoid repetition on the keyboard. He would start the system up and specify the command he wants, choosing either a circular movement or a swipe to represent the `ssh` command. Later, instead of spending time typing the same command for seconds and seconds, he can simply swipe or draw a circle and receive instantaneous feedback and execution.

How the system works

Architecturally, GesTerm depends on several bridges: the macOS-Python, Leap-Python, and Python-terminal bridges. These come together to give the user a seamless experience after running the gesterm.py script. Leap Motion gives Python (GesTerm, really) frame data to analyze, the Python script detects the gesture performed, talking to macOS for user alerts and executing the command in the terminal.

GesTern works well enough when supporting the two gestures mentioned previously. There is enough of a difference between the finger movements associated with circles and swipes (in any direction) that GesTerm can pick them up without any changes to the default thresholds configured in the Leap Motion sensor itself. Even so, because of the dynamic nature of gestures in general, there is always the risk of misdetection. Notably, a swipe that is *just slightly curved* may be detected as the start of a circle movement, and this, of course, leads to unexpected system behavior. Furthermore, on the execution side of things, once the proper Python library was found, keystroke generation was trivial, with response times being consistently under a second.

I initially tried to incorporate taps, circles, and swipes together to give the user more freedom, but this required much configuration of the default thresholds at which these gestures

would be detected. It is easy to work with GesTerm when gestures are isolated, but as a whole, everything becomes difficult to deal with. Paired with the Leap Motion sensor's varied performance under different settings—my dorm room, lit by only small warm bulbs, and 24-121, the demo room lit by ambient window light and LEDs from above—this mess of configuration becomes even more complicated. Because of these issues, I simplified my model down to what was described in the previous paragraph, opting to support just circles and swipes. Perhaps a suitable solution would involve environment detection in real time and automatic adjustment of the gesture thresholds to compensate.

Roadblocks and modifications

In my initial proposal and pitch, I wanted GesTerm to support the four default Leap Motion gestures (circle, swipe, key tap, screen tap) *and* custom gestures trained through machine learning. I believed I was prepared for the machine learning component of the project and ported the Gesture Recognition Toolkit to Python 3.7 ahead of time. When it came time to actually experiment with gesture data, I realized that there were far too many issues to address alone:

1. What kind of model was I looking for? An online, streaming model for fast iteration with different users seemed ideal, but I couldn't see how to implement this idea concretely. I did refer to our previous lectures regarding an online penstroke recognition model to come up with this idea.
2. How would I separate gestures from each other, and what kinds of transformations would I have to do to the time series data to make a proper model? Given my lack of real background in machine learning (I have taken neither 6.036, Intro to ML, nor 6.867, ML), I figured that spending time on machine learning rather than the core goal, which was to give the user a better command line experience, was infeasible.

3. What overall pipeline did I want to support custom gestures? The answer relies on those to the previous two, and thus, I had no clear vision.

Further, as I mentioned in the previous section, **System overview: How the system works**, the issue of suitable gesture detection thresholds required me to support only two gestures for this iteration of the project.

Feedback incorporation

One classmate repeatedly encouraged me to better the user experience by actually informing the user of when a gesture is started and completed. I was daunted by the idea of implementing user alerts, but after discovering the API for calling macOS notifications through Python, this became a crucial component of the user experience. If I had not fixed this, GesTerm users would essentially be gesturing in the dark, not knowing whether their movements are detected at all. The alerts also integrate with the operating system, so there is negligible intrusion.

User studies

I did not conduct actual user studies during the course of the final project season. I detail below a user study design that I believe would work to reveal GesTerm's highlights and shortcomings.

The goal

One question comes to mind: *can my users naturally, reliably, and accurately make use of gestural macros without needing to think about it?* I want them to feel like the gestures they perform are just as familiar as an old friend.

The philosophy of the study

Based on the Norman-Neilson Group's article regarding user studies, I decided that my studies will be 1) behavioral, 2) qualitative, and 3) strategic, according to the three dimensions and their

levels. We want to see what people do with GesTerm, their holistic thoughts on how the system integrates itself with their existing keyboard/touch modality, and whether there are new ideas and suggestions for us to better the product and provide a better user experience. In terms of the assignment handout, I want this study to focus on how well users carry out a specified task.

Target audience

I want to expose a variety of command line users to GesTerm. I plan on taking data from roughly 25 people ranging from new computer science students here at MIT to creative hackers who have optimized their terminals for maximum keyboard productivity. Exposing GesTerm to such a wide range of subjects will give me a better idea as to where within the current ecosystem GesTerm fits.

The task

I will ask users to launch GesTerm with the same format in **System overview**. Users will specify their own commands to save as macros. I will give them three minutes to play around with the system and reinitialize GesTerm if they so wish. The goal is to let the user experiment with macros in a natural way by using their own commands, not by forcing them to follow commands I want them to use.

Data to be recorded

I plan on taking both qualitative and quantitative measurements.

For the qualitative, I will ask users for their opinions on how seamless GesTerm is with their current experience with the command line and, on a scale of 1 to 5, how much more productive they think GesTerm made them. On the same scale, I will ask how learnable the system was.

For the quantitative, I will, assuming I am given permission, record/quantify users' face expressions during their sessions and deriving a general vibe from them. Any emotions visible on

the face, such as disgust or joy, will indicate GesTerm's effectiveness. I expect such emotions to occur most right after GesTerm is invoked. Existing solutions for affect detection that we have covered in class can be used for this approach.

Script

Please look at me as I launch GesTerm from Python on my own computer…

Before we begin, do I have your permission to record your face expressions? We will be using this data to quantitatively measure how pleased you are. It is completely optional.

To review, you can perform your commands by either swiping or drawing a circle above the Leap Motion sensor here.

Please launch GesTerm and specify your own gestural macros. You will have three minutes to play with the system as you please. Try and experiment with different scenarios where you might like to use your saved macros!

How seamlessly do you think GesTerm integrates with your experience with the command line?

On a scale of 1 to 5, how much more *productive did you feel using GesTerm?*

On a scale of 1 to 5, how learnable and natural was GesTerm for you?

Overall, what did you like and dislike about GesTerm?

Thank you! We hope you enjoyed using GesTerm and that it was a productive experience for you.

Performance

While GesTerm may not have worked perfectly, for the reasons and shortcomings I detailed in **Roadblocks and modifications**, I learned a lot about the nuances of these processes. Working with gesture data through infrared LEDs and two low-resolution cameras is not always perfect: environments can introduce noise, and when it is the hardware itself that determines how a time

series should be parsed as a gesture, beyond threshold configurations, there is not much we can do as designers. As I described previously, to improve the system, I would add automated environment calibration and fully build in machine learning for custom gestures to truly make the system state-of-the-art and more intuitive. I hope to see GesTerm progress beyond just circles and swipes, to progress to the level of immersion that modern UIs introduce, such as the grounded-in-reality "trash can" that feels so right to computer users who delete files and folders.

Packages/libraries used

GesTerm's code repository can be found at this link on my personal Github account.

I assume that GesTerm will be run on macOS-based machines or similar hardware that runs macOS (Hackintosh). No support has been added for Windows 7, 8, or 10.

To build GesTerm, I used a variety of packages and libraries:

1. Python 3.7, which served as the entire platform on which GesTerm was built. Python can be found on the Python Software Foundation's download page here.
2. Leap Motion Python SDK v3.2. This SDK was ported over to Python 3.7 for this project and can be found ready for use in my repository.
3. Gesture Recognition Toolkit. This was downloaded from the course website and then ported over to Python 3.7, the instructions for which are in my repository. I did not end up using GRT, as the machine learning component of my project was too ambitious in scope.
4. `pykeyboard`/`pynput`. These libraries served as the keystroke generators for the project.

To get my system running, please install the dependencies through

```
pip3 -r requirements.txt
```

and initialize the system with the same command format as presented earlier in this document in **System overview**.

The ported libraries should work as downloaded, and GesTerm will function normally provided a Leap Motion sensor is present.

Acknowledgements

Thank you to the course staff for introducing me to an entirely new area of computer science that I had not previously thought or known much about. I am glad to have been in the class, and I hope to incorporate multimodal user interface design in my future work in computer systems.

Thank you to Professor Davis, Carolyn Lu, and Mary Thielking!