CS50 Final Project:
Flying the Crazyflie Using a Leap Motion Controller

by Yondon Fu and Patrick Xu

***Abstract**: This document outlines our design for the CS50 final project for which we programmed a Crazyflie nanocopter to fly using a Leap motion controller. In this report we describe our threaded design, data structures, functions, project extension, and the lessons learned .*

Introduction

In this project we utilize the Leap API and the Crazyflie nanocopter API to write a program that takes input from the Leap motion sensor to fly the nanocopter. We use a threaded design to facilitate communication between the Leap device and the Crazyflie nanocopter.

Threaded Design

The **main thread** creates two threads:

The **leap controller thread** detects hand gestures, processes them and either sends out appropriate state change signals to the copter or adjusts the copter's pitch, roll, yaw and thrust. Callbacks are used to continuously get hand data from the Leap device.

The **copter controller thread** changes the copter's state based on the signal sent from the leap controller thread and sends updated pitch, roll, yaw and thrust information to the copter. This flight information is sent through packets using the Crazyflie radio. It then calls appropriate control commands (fly, hover or land) to the copter based on its current state. This thread can also read log information back from the copter, allowing the user to utilize sensor data.

Global variables are used to keep track of the copter's current state, signal, thrust, yaw, roll and pitch. We make sure our global data is "safe" with a mutex lock. Whenever one thread is running, we lock the global data so the other thread cannot access or alter it until the current thread is done running.

Program Design and Specs

Gestures

The gestures recognized by the Leap at any given time vary depending on the current state. The Leap recognizes when there are no hands, one hand, and two hands in its field of vision. There is generally a main hand, which controls the flying parameters (thrust, pitch, roll, and yaw) and a secondary hand, which changes states. Therefore, the gesture "One hand" refers to the the user using the dominant hand only while the gesture "One hand + two fingers" refers to using the main hand as well as the secondary hand.

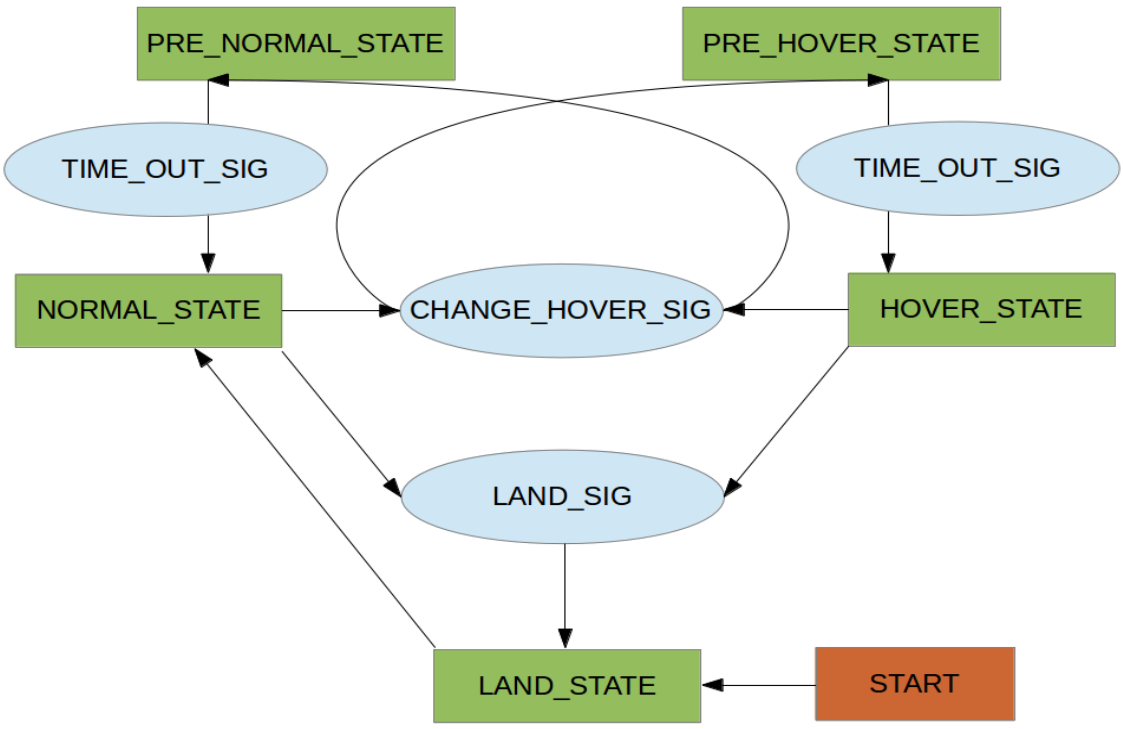
State	**Gesture**	**Result**
NORMAL_STATE	No hands	Change to Land
	One hand	Normal Flying
	One hand + 2 fingers	Change to Hover
HOVER_STATE	No hands	Hover with no pitch/roll

	One hand	Normal Hover
	One hand + 2 fingers	Change to Normal
	One hand + fist	Change to Land
LAND_STATE	No Hands	Stay in Land
	One hand	Change to Normal

Note that the user should keep the fingers of the main hand spread. This increases the accuracy of the reading by the Leap.

Finite State Machine (FSM)

Real-time control of our copter is made possible using a FSM. The computational model helps us control what actions the copter should perform based on what state it is in. Signals are used to tell the copter when to change states. The copter then calls a specific function depending on the state. Our FSM is diagrammed below.



When the copter is in NORMAL mode, we can control its thrust, pitch and roll. When the copter is in HOVER mode, we can control its pitch, roll, and yaw. The thrust will be set at a constant "hover" value that signals the copter to hold its current altitude. As a result, we are able to fly the copter around without needing to pay attention to the altitude. When

the copter is in LAND mode, we can no longer change its pitch and roll. Furthermore, the copter will gradually decrease its thrust until it reaches the original starting thrust, thus lowering the copter softly to the ground. See flight control for an explanation on how use hand gestures to control these flight parameters.

We use PRE_NORMAL and PRE_HOVER as transitional states to prevent accidentally switching between multiple states with one gesture. When we send the CHANGE_HOVER signal the copter changes to one of these transitional states. In these transitional states, the copter behaves as if it is already in the next state. For example, after switching from NORMAL_STATE to PRE_HOVER_STATE, the copter already begins to hover. Thus, to the user it feels like there is no transitional state. The copter switches to the actual NORMAL or HOVER state when it receives a TIME_OUT signal, which is sent out periodically by the leap controller thread.

To ensure that the leap controller thread only runs after the previous signal has been processed, the thread only runs if the current signal is NO_SIG the copter controller thread sets the current signal back to NO_SIG after it is done sending control commands to the copter.

Data Structures

The main data structure we use in our program is the CCrazyflie structure. We are constantly updating the pitch, roll and thrust of the structure based on the gestures read from the Leap device.

The HOVER state of our copter is relies on one of the fields of the CCrazyflie structure, m_setHoverPoint, and certain functions defined in CCrazyflie.cpp. When we want the copter to hover, we set m_setHoverPoint to 2 in the control.cpp file. The copter controller thread is constantly calling the cycle() function on the CCrazyflie structure so when that function detects that m_setHoverPoint is equal to 2, it calls sendParam() with an althold value of 1. This tells the copter that it is in HOVER mode. Afterwards, we set m_setHoverPoint to 1 to make sure we do not accidently call sendParam() again. If m_setHoverPoint is equal to -2, cycle() calls sendParam() with an althold value of 0 which tells the copter that it is not in HOVER mode. Afterwards we set m_setHoverPoint to -1 to make sure we do not accidently call sendParam() again.

The other data structure we use is the CCrazyRadio structure. We initialize this structure to begin communication with the copter.

Fight Controls

The thrust of the Crazyflie is controlled by the height of the hand. As the hand gets farther away from the Leap, the thrust increases. The user will be warned when the hand is getting so low or so high that the Leap may lose track of the hand.

The pitch of the hand is controlled by leaning the hand forward or back. Think of dribbling a basketball. As the fingers dip below the horizontal plane of the wrist, the copter will fly forward. When the fingers are above the wrist, the copter will fly backward.

The roll of the hand is controlled by keeping the hand in the same horizontal plane as the wrist but moving the hand side to side. If the fingers are to the left of the wrist, the

copter will fly to the left. If the fingers are to the right of the wrist, the copter will fly to the right.

Yaw control depends on the height of the hand, taking the place of thrust control, and thus is only active when the copter is in hover mode. When the height of the hand is near the top of the height range the copter rotates clockwise. When the height of the hand is near the bottom of the height range the copter rotates counterclockwise. The copter does not rotate when the hand is in the middle of these two areas. Clockwise and counterclockwise are in reference to the copters rotation when viewed from above.

The user should always know in which direction motor 1 (M1) is facing. All of the above controls are from the perspective where M1 is facing away from the user.

Function Decomposition

Our code uses certain helper functions within the two threads. The comments in the control.cpp file describe how each function works.

Extensions

Variable Pitch, Roll, and Thrust

When the Leap thread calculates the flight parameters pitch and roll it uses the hand position to calculate the magnitude. For the hand gestures used by pitch and roll, the Leap API reads the hand as a position and returns a value between 1 and -1. We multiple this value by a predefined number by this value. This allows us to have use fine-grained controls to fly the copter.

The process for calculating the thrust is quite similar. There is a range of height values that the Leap thread takes as valid. This range of values falls just inside the maximum range for the Leap sensor. A value is calculated by dividing the height of the hand by the maximum height. This value, which falls between 0 and 1, is then factored into an equation that returns a number ranging from the minimum allowed thrust to the maximum allowed thrust. The equation is

$$T = \frac{height - HEIGHTmin}{HEIGHTmax} * (THRUSTmax - THRUSTmin) + THRUSTmin$$

where T is the final thrust sent to the copter. This does not take into consideration the battery level of the copter. We found that overall performance is much more predictable when the user knows exactly how much thrust given to the copter.

Yaw Control

The specifics of yaw control is outlined in the section Flight Controls. The main reason for having yaw controls available to the user is so the user is able to correct the orientation of the copter. If M1 is not facing the user, flying becomes very difficult. If this occurs, the user is able to correct the orientation (in hover mode) and continue flying.

Keyboard Control Extension

In this part of our extension, we programmed the copter so we could fly it using keyboard commands. The idea behind this was that we believed that the keyboard could provide smoother flight than the not always reliable gesture detection of the Leap device. To accomplish keyboard control we used a threaded design similar to the threaded design for our baseline.

The **main thread** once again creates two threads:

The **keyboard controller thread** takes in a command from the user. The command is processed by a helper function that determines the state, thrust, pitch and roll of the copter. This is much simpler than the design for the baseline because we no longer need to use signals.

The **copter controller thread** sends updated pitch, roll and thrust information to the copter. It then calls appropriate control commands based on the copter's current state.

Once again we use a mutex lock to ensure our global data is safe. Although we do not use the state-change signals from the baseline, we still use the current signal variable to switch between the PROCESSED and NEEDS_PROCESS signals. These signals help ensure that the keyboard thread only runs after the previous command has been processed and that the copter thread only runs when there is a command that needs to be processed.

We did not have time to implement truly real-time keyboard controls so instead we coded our program such that a user-inputted command will be run continuously until the user manually stops it. To accomplish this, we used CTRL-Z as the command to stop running a command. We wrote a signal handler that catches the CTRL-Z signal and tells the copter thread to stop calling control commands instead of stopping the program completely (we switch current signal from NEED_PROCESS to PROCESSED). Afterwards, the user can input a new command. To truly terminate the program the user must use CTRL-C.

Note: when using the keyboard, the copter can only fly forward, backward, left or right in a straight line.

When flying the copter using keyboard control the user must input a command, let it run until he or she is satisfied, CTRL-Z out of the command, and then input a new command.

We realize this method of piloting the copter is not incredibly efficient, but given our time constraints, this is what we decided to implement. If we had more time we would definitely look into coding truly real-time controls with the ability to dynamically change the pitch and roll. Going beyond that, we would have liked to allow the user to type in a string of words and the copter would process those commands in succession. Another cool possibility would have been to integrate a speech to text API into our program to control the copter using verbal commands.

Lesson Learned

This project offered many challenges. Some of these, such as having to deal with a real time system and using threads, were known to us when we started the project. Many other challenges were realized along the way. Up to this point in the course, we only had dealt with software. It turns out that hardware can be just as "buggy" that software. Broken motor mounts, unbalanced propellers, drifting while in hover mode, and finicky gesture recognition are just some of the issues we had to overcome in our short, one week development period.

A lesson we learned during this project is that it is not always possible to run your code again and again. As shown by the picture to the right, hardware, unlike software, has a limited lifespan. In this case, after repeatedly testing the hover functionality of our nanocopter (and repeatedly slamming the nanocopter into the ceiling of the common room) the plastic motor mount broke. In other labs, code could be run time and time again to test and debug. Even in the Crawler lab when we handled large amounts of data, we could simply crawl at depth 1 and a run would be done in seconds.

We quickly realized that we had to take precautions when running code. If the code was especially buggy, one of us held the copter in place while the other tested. Only when we were at least moderately sure of success did we allow the copter to actually fly.

In the end, the plastic motor mount ended up being an easy fix. With the spare motor mount provided, we were able completely repair the copter in less than 20 minutes. However, it is a good thing we learned our lesson with something that was so easily repaired. Had an actual motor broken, or any number of sensors on the nanocopter itself, we would have been in much greater trouble. Hardware is different than software, and as such, must we treated differently.