Robot Control through Gesture Recognition

Ben Yett
Charlie Hartsell

Introduction

Nearly everyone interacts with some type of computer on a daily basis, primarily through traditional user input methods such as a keyboard and mouse. However, as computer-controlled systems have become more prevalent in everyday life, there has been a significant need for more flexible and intuitive means of gathering user input. In response to this need, much research has been done in the area of Human-Computer Interaction (HCI) which, among many other things, has led to commercially available devices such as the Microsoft Kinect sensor system. Our project utilizes this sensor to create an intuitive and easy-to-use robot control system where physical gestures are the primary means of user input. In particular, the user is able to control a simulated iRobot Create with a variety of simple arm motions.

Approach

Our system is divided into three primary subsystems: gesture recognition with the Kinect sensor for providing user input, a finite state machine (FSM) for robot control logic, and the Gazebo simulator and corresponding interface for modeling the physical robot. The system was designed to be easily ported from controlling a simulated robot to a physical robot. With this consideration in mind, the simulation and interface subsystem was constructed as a separate entity from the gesture recognition and FSM control logic, which are contained within a single program referred to collectively as the Robot Controller. Additionally, the Gazebo simulation and interface programs were restricted to run in a UNIX environment due to no suitable Windows release of Gazebo being available. Because of similar issues with the Microsoft Kinect SDK, the Robot Controller program was restricted to run in a Windows environment, and a diagram of the overall system layout is given in the figure below. For these two separate parts of the system to communicate, a two-way TCP connection was chosen for sending robot commands and any other necessary system overhead due to the reliable nature of TCP. An additional one-way UDP connection was used for transmitting sensor data from the Gazebo interface to the Robot Controller program since an occasional dropped packet was an acceptable cost for the reduced latency and overhead of UDP relative to TCP. This modular design means that only the interface between the Robot Controller and Gazebo would need to be modified to support a physical robot instead of a simulated one. Additionally, many of the embedded processors commonly available on physical robots may not offer sufficient computational power for the image processing tasks associated with the gesture recognition subsystem, necessitating that this subsystem be run on a desktop-class computer. Also, there is no requirement for the two separate sections to be run on independent computers, but instead they can be run using virtual machines to provide the necessary operating system environments.

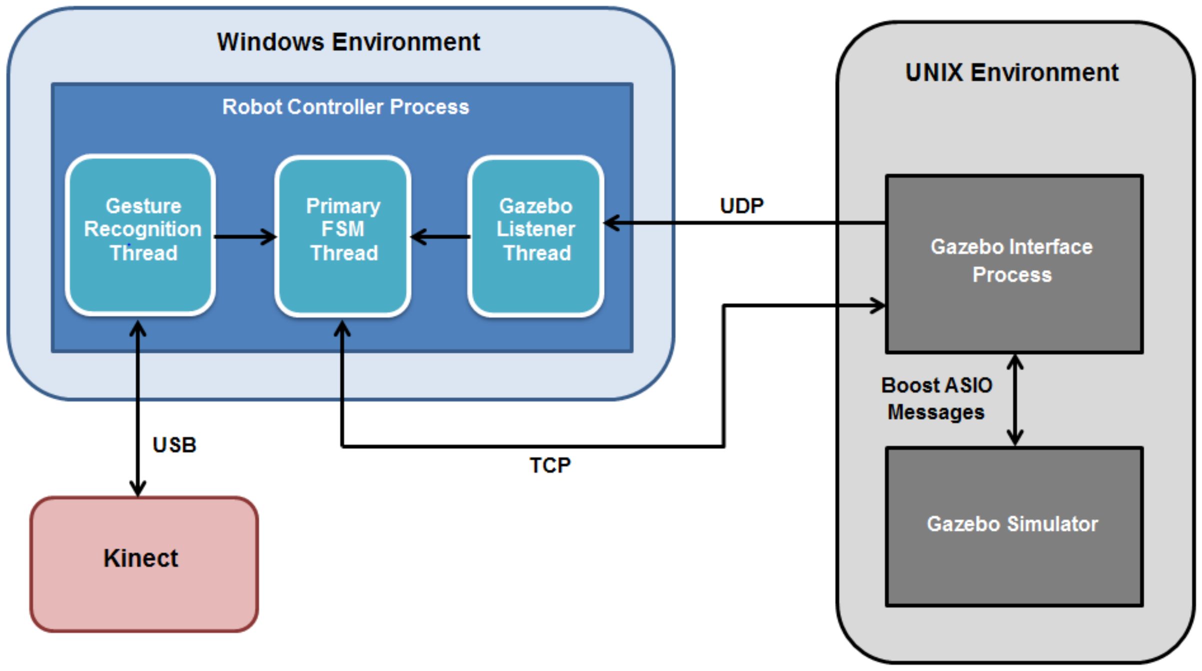

Figure 1. Overall system layout.

An accurate, low-latency gesture recognition subsystem is a necessary component of the overall system and is crucial for a positive user experience. Our team was restricted to using a Microsoft Kinect due to the availability of both the hardware as well as a no-cost, publicly available SDK. Due to our team's inexperience with gesture recognition and the desire to keep the system easily portable to various hardware, the gesture recognition itself is relatively simple. We start by assigning the proper element of the skeleton.SkeletonPositions struct to a variable of type Vector4& (essentially the x, y, and z positions of the indicated body part). Seven body parts were needed: the right and left hands, the right and left hips, the right and left elbows, and the right shoulder. We also have two variables to keep track of the backwards gesture (backwarda and backwardb), two to keep track of the forwards gesture (forwarda and forwardb), two for the autonomous mode gesture (autoa and autob), two for manual mode gesture (mana and manb), one called stop for the stopping gesture, and finally two called turncount and result related to the turning gesture. The first thing checked after this is whether the right and left hands are both above their respective hips - the idea is for the user to turn by holding an imaginary steering wheel, which this position tries to emulate. If so, and assuming the hands are not at the exact same y position, we clear all variables not related to turning and increment turncount. Turncount was mostly used earlier in our gesture recognition testing to prevent extremely rapid angle measurement outputs, but has since been rendered largely unnecessary as we discovered the simulated robot was not responsive enough to turning. To prevent future divide by zero problems, we check to see if the right and left hands are in the same x positions (such as turning the wheel sharply in one direction with one hand directly above the other). If so, we set the angle measurement to $\pm\pi/2$ as appropriate (turning right is negative, left is positive). Otherwise, we take the x and y positions of both hands in order to geometrically calculate the angle with atan. Then, if turncount has reached the proper value (1 currently), result is divided by turncount, assigned to user_arg (used to send the angle to the robot), and used to determine if a right or left turn is occurring so that the proper CMD_MASK is set before clearing all variables.

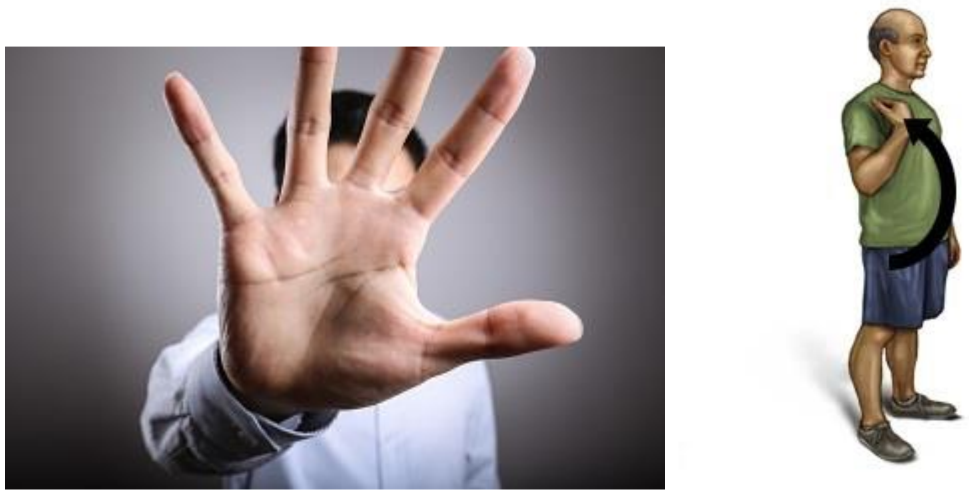

Figure 2. Examples of a few of our gestures: stop and forward [1], [2]

If no turn was attempted in the previous step, our input to the robot is the STOP_TURN_CMD_MASK, which will tell the robot to stop turning. Otherwise, the robot would continue to turn even after the user stopped applying turn commands; this was not the behavior we were intending. The next possible gesture is to stop. A single stop gesture is seen in every frame that the user has his or her right hand more than 0.15 meters above the right shoulder - like a hand raised, palm out, that normally signifies stop. If this is the case, we clear a few other variables and check the current stop variable value. A stop value of five currently corresponds to sending a stop input to the robot; this value has changed over time depending on how quickly we wanted it to stop while also trying to prevent accidental commands. All variables are also cleared upon recognizing that a stop input is being sent. On the other hand, if stop is currently less than or equal to four, we simply increment stop. Next, we have our forward and backward commands. The above image gives a good example of making the forward motion; start with the right hand below the right elbow and then raise it above the right elbow. To actually send a forward command to the robot, this process must happen twice before any other gestures are performed that interrupt the sequence. The forwarda and forwardb variables are basically just used to ensure that it does happen twice. After completion of the second motion, the FORWARD_CMD_MASK is set, and along with that all of the variables are cleared. Sending a backward command is a similar process. It instead uses the left arm, and the motion begins when the left hand is above the left elbow. The left hand is then lowered below the elbow to complete one gesture, and again two complete motions are required to send a command to the robot. Essentially, this is the opposite of the forward gesture, and uses the opposite arm to avoid confusion. The processes to recognize gestures corresponding to autonomous and manual commands are also similar to each other. These are both essentially waving motions, which in code matches up with the x position of the hand switching between being greater than or less than the x position of the respective elbow. In addition, the hand must have a y position larger than the y position of the elbow. The autonomous gesture is with the left arm, and the left hand must start to the left of the elbow to begin the motion. The hand must then be moved to the right of the elbow to complete the motion, and again two complete motions must occur in order to send the robot an autonomous command. A manual command is initiated with a similar procedure, with the right arm being used and the hand being both to the right of and above the elbow to begin the motion. When either of these gestures are recognized, all variables are cleared after the input to the robot is appropriately set.

```
if (rh.y > (rs.y + 0.15)) {
        backwarda = 0;
        backwardb = 0;
        if (stop == 5) {
                clearall();
                user_input |= STOP_CMD_MASK;
        }
        else if (stop <= 4) {
                stop++;
        }
}
```

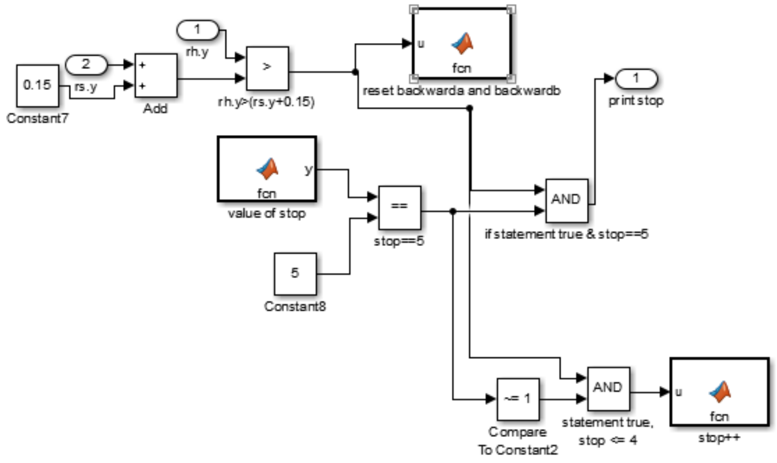

Figure 3. Comparing the C++ code and Simulink implementation of the stop function

Another major component of our project involved using the MATLAB toolsets Simulink and Stateflow. In Simulink, we were able to completely recreate the gesture recognition program that is used to provide commands to the robot. It starts out with generators for the system that mimic the x and y positions of the appropriate body parts - right hand, left elbow, etc. This has been tested with constants, custom built waveforms, and values randomly generated at each time step. These values are then used as the appropriate inputs to the various subsystems that replicate the process used to determine if a gesture has been recognized. As a simple example, the stop subsystem receives values of the y positions of both the right hand and shoulder. These inputs then go through various boxes such as logical operators and comparators for refinement, and can also change the variables that are used identically to the original program. The variables use the Data Store Memory type to maintain their values throughout the model, and are modified using MATLAB function blocks. Function blocks are also used to read the current variable value as necessary for the various gestures. The final output for each subsystem is either a single binary value for a recognized gesture or, only for the turning system, a real value representing the current turning angle plus a binary value returning a zero when the vehicle is supposed to be turning or a one to signal to the robot to stop turning when no turn gesture is present. These outputs are used for two tasks. The first is as inputs to one large OR block, such that when any gesture is recognized, all of the Data Store Memory variables are cleared. A second use for the results is as inputs to the Finite State Machine (FSM) that is implemented using Stateflow. The subsystem values are joined by three other inputs that represent the wall (front) sensor along with the right and left tilt sensors. Currently these values are defined by simple pulse generators to simulate the various obstacles.

At this point, our model will have generated body positions, processed the data, and determined the currently active gesture if one exists. All that is left is to use the recognized gestures as appropriate guard conditions between the states of the FSM. Our FSM always begins with the robot in the most basic state - Manual mode, within the Stop block, and completely stopped (no turning). If no gestures are ever recognized or obstacles ever detected, the robot will remain in this state for the full length of the program. Otherwise, the system will react appropriately and change states as required. The possible transitions from this initial state are: to the TurnL and TurnR states also within the Stop block upon a turning angle input, to the Forward and Backward blocks and states upon a forward or backward input, to the Avoid_Obstacle state upon a trigger from any of the available sensors, to the initial (Forward) state of the Autonomous mode assuming an auto input along with the lack of any obstacles, or to the Backward state of Autonomous mode when receiving an auto input while a sensor is triggered. These two separate transitions are necessary in order to avoid the arrival into an unsafe state. The only safe states in the FSM while a wall or other object is triggering a sensor are the Avoid_Obstacle and Stop states in Manual mode along with the Backward, TurnL, and TurnR states within the Autonomous system.Therefore, if the transition from manual to autonomous is enabled while an obstacle is also present, there would be a brief period where the FSM would be in an unsafe state due to the fact that the initial state of Autonomous mode is unsafe. The FSM as coded for the robot is able to account for this using hierarchy - after switching from manual to autonomous it checks for the presence of obstacles before the system ever outputs the switch. Therefore, this accommodation better mimics the real model. Another consideration during operation of the robot is liveness constraints. The only state where no actions are being taken by the robot is the stop state. While it is necessary to have as an initial state and can be considered safe, transitions should be made to other states as soon as possible, either when a gesture is recognized or when an obstacle is present so that steps can be taken to avoid the obstacle.

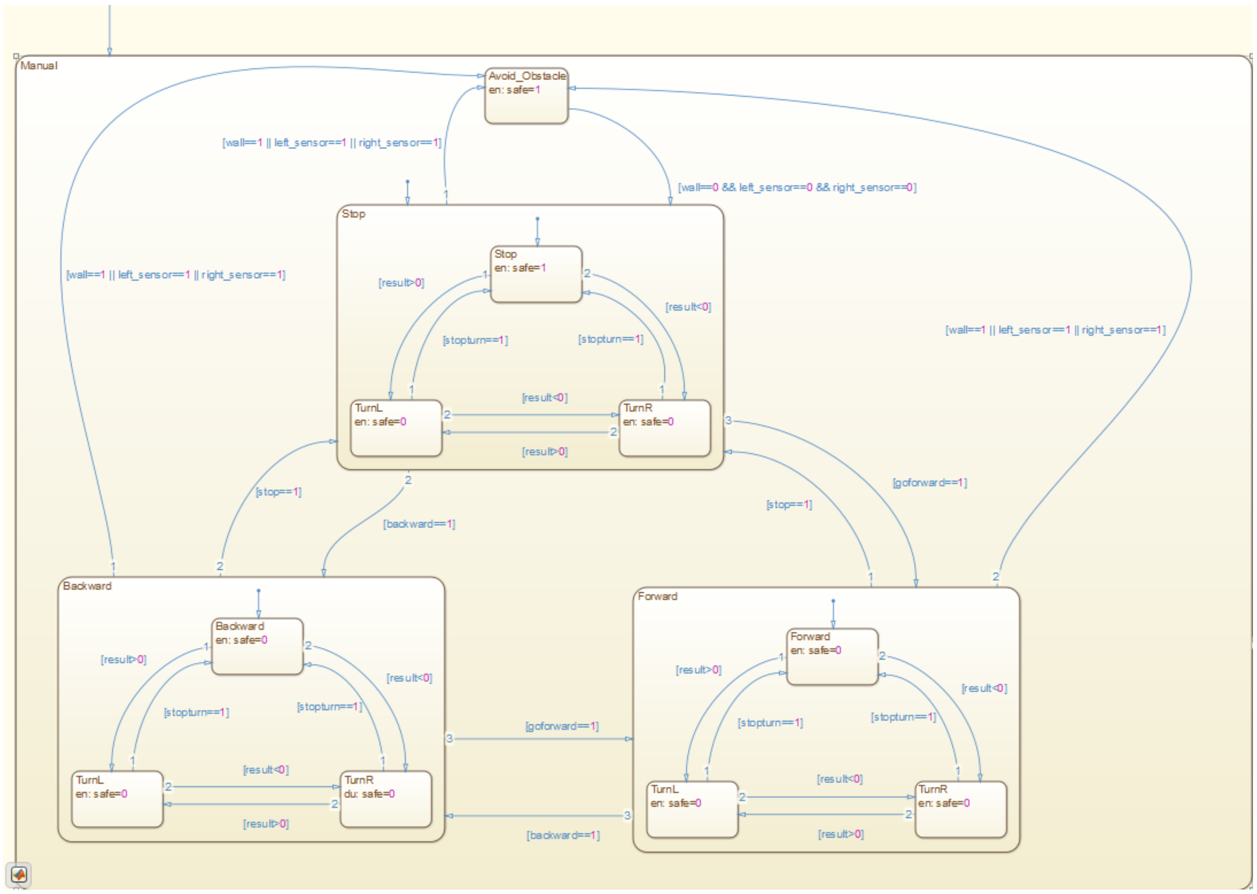

Figure 4. Manual mode in our Stateflow FSM

Challenges

Properly interfacing with and utilizing the Gazebo simulation environment was a significant technical challenge encountered during the project. In particular, sufficient documentation regarding how to send control messages to robots using differential-drive type steering was not available, and determining these technical details with the limited documentation available required a significant amount of time. Difficulties with Gazebo were an expected obstacle for our project, and were identified as a potential risk in the project proposal along with a possible mitigation strategy of utilizing the existing Gazebo model libraries and environment creator. The online model libraries provided a model of an iRobot Create which needed only minor adjustments for our project, and a world model from a previous project provided a suitable testing environment. Using these pre-made models instead of creating original models saved a significant amount of time, allowing for more time to be spent resolving technical issues with Gazebo.

Gazebo also presented two significant limitations we were unable to overcome, but that were deemed acceptable to continue using the software. The physics simulation provided by Gazebo was not as accurate as expected and resulted in unrealistic behavior of the robot in certain situations. For instance, when the robot impacted a wall, there was a significant chance that it would drive vertically up the wall until it reached a tipping point and flipped itself over. This is entirely unexpected of a real iRobot Create where impacting a wall would simply bring the robot to a stop. We determined that this issue was only a slight inconvenience and in rare cases would result in a simulation run being invalid. Additionally, Gazebo appears to impart a significant latency when issuing motor commands from the robot interface into the Gazebo simulation. This additional latency resulted in a user-noticeable delay between performing a gesture and the robot responding to the gesture, and its exact cause is unknown. Our team attempted to quantify this latency experimentally by running the robot controller and the simulation on a single computer while using the Time Stamp Counter (TSC) register as a synchronized clock source. However, the Gazebo simulation is restricted to running in a Linux virtual machine due to compatibility issues, and it was discovered that the virtual machine used a virtual TSC instead of the true physical TSC onboard the processor. We decided not to invest the additional time required to quantify and troubleshoot this latency since it was only a minor inconvenience to the user, and instead spent this time troubleshooting other aspects of the system.

The Kinect sensor presented another set of challenges, particularly due to our team's inexperience using the device and its SDK. This was another major unknown with our project and was identified as a source of risk in the proposal. The learning curve associated with using Kinect required a significant time investment, but was not beyond our original expectations. However, we encountered one particularly troublesome issue where the sensor would occasionally detect and lock in on background objects as well as the primary user. This occurred at seemingly random intervals, and would result in the program failing to detect user input without reporting any error condition. Additionally, the SDK tools we used did not visually indicate that another object was being tracked when testing using the provided examples. Once the cause of the problem was determined, a simple check to determine which of the tracked objects was the primary user was implemented and resolved the issue.

Another interesting challenge came through attempting to design and implement the FSM. It needed to have reliable and repeatable transitions between states while preventing us from reaching any unsafe states if an obstacle was present. Due to lack of experience with Stateflow and the difficulties of initially visualizing all possible paths, some changes needed to be made to correct errors that would have otherwise been present. One of these has already been discussed - we originally did not consider that a switch from autonomous to manual mode could occur while an obstacle was present, resulting in the necessary conditions added on those transitions. We also had a hard time determining whether the system was always in a safe state until adding in the safe variable that is set whenever a safe state is entered and cleared whenever an unsafe state is entered. Cross-referencing this with whether or not a sensor is recognizing an obstacle turned out to be a very effective means of testing. All available software for formal verification and validation of a Stateflow model seemed to cost money, so we needed things like this to accommodate and adapt as best as we could. Other problems arose from not understanding the hierarchy of Stateflow, with transitions inside certain layers of blocks not triggering when we wanted them to due to a transition of higher priority taking its place. With Simulink, the main concern was not being able to create a sufficient series of tests to ensure safety and liveness requirements. A few methods were tried, such as randomly generated values, specific constants, PWM generators, and handcrafted waveforms. Using some combination of these seemed to be enough, but we were unable to discover a useful means of creating a larger test set for the system to run through - everything needed to be done by hand.

We also ran into some issues when it came to determining the best way to implement the recognition process for each gesture. The initial plan was to rely on completed research for the best means of setting up values and keeping everything different enough to prevent crossover; however, there was just not as much relevant information available as we had expected. Therefore, a good amount of time was spent fine tuning the parameters such that gestures could be recognized properly and that incorrect gestures were not recognized. It was difficult to implement each of the gesture possibilities on just two arms. Originally, the stop command kept occurring while trying to tell the robot to move forward - the additional distance above the shoulder and the necessity of several consecutive frames at that position were accommodations for this. Additionally, there was some overlap between the forward/manual gestures and between the backward/auto gestures. While the chance for either to happen is still present, some of that was mitigated by forcing the hand to remain above the elbow throughout the waving process. Clearing all of the variables when any gesture is recognized and clearing certain ones at other points in the process were concessions to overlap that we were experiencing. The most difficult gesture to properly enable without causing issues elsewhere is definitely the turning process, as two hands are required for that as opposed to one for the other gestures. Requiring both hands to be above the hips solved some issues, but others still seem to be present. Even after solving the majority of these problems, even the two of us did not have the gestures fully under control, as could be seen during our demonstration. More fine-tuning would still be needed for this part of the project moving forward, and sufficient explanations would need to be given to any potential users for them to perform gestures properly. However, a lot of progress has been made, and even with a few miscues we were able to show that the gesture recognition aspects worked sufficiently well to control the robot.

Results and Future Work

To quantify the accuracy of the gesture recognition subsystem, a test was performed wherein a user performed various input gestures in front of the Kinect sensor and the output from the gesture recognition software was recorded. The test data is given in the table below with the correct system responses highlighted in green. The results show a typical gesture recognition accuracy of 90% or greater depending on the particular gesture. It is also notable that there are only a few cases where an input was provided with no gesture being recognized and vice-versa. Additionally, the ‘Forward’ and ‘Manual Mode’ inputs appear to be the most difficult for the system to recognize with the lowest accuracy ratings of 90 and 91.3% respectively. It should be noted that this data was collected with a team member providing input while standing within the ideal operational range (4-11.5 ft) of the Kinect sensor. The recognition accuracy would likely decrease with a less experienced user providing input or if the user stood in the non-ideal range of operation.

Table 1. Accuracy of gesture recognition software.

Output / Input	Forward	Backward	Stop	Turn Left	Turn Right	Auto	Manual	No Output	Totals	Correct Prediction (%)
Forward	18	0	0	0	0	0	0	2	20	90.0
Backward	0	21	0	0	0	0	0	0	21	100
Stop	0	0	20	0	0	0	0	0	20	100
Turn Left	0	0	0	20	1	0	0	0	21	95.2
Turn Right	0	0	0	1	20	0	0	0	21	95.2
Auto	0	1	0	0	0	20	0	0	21	95.2
Manual	2	0	0	0	0	0	21	0	23	91.3
No Input	0	1	0	0	0	0	0	0	1	-

Our team achieved our primary goal of controlling a simulated iRobot Create using gesture recognition as the primary source of user input. In addition, all deliverables were submitted by their respective due dates including the presentation and demonstration, the software package, and this final report. No major design changes were required from the design outlined in the proposal. Project development fell behind the originally proposed schedule by approximately one week, but this did not result in any missed deadlines. Our team was unable to achieve the stretch goal of implementing the control system with a real robot due to both the unavailability of an iRobot Create as well as insufficient time to implement and troubleshoot the additional hardware and software required.

Our project leaves significant opportunities for future work, with the most obvious being to achieve our original stretch goal of implementing the system with a real robot. This would require porting the robot interface to an embedded board compatible with the iRobot Create. Since all communication between the controller and the robot interface occurs over standard network sockets, a Wi-Fi capable embedded board should make this a relatively straightforward task. Additionally, reconfiguring the current SITL testing environment to use a different simulation environment could resolve the physical accuracy and command latency issues present with Gazebo. The current system could also be expanded with additional gestures to support more complex robots with additional functionality such as gripper arms. This would require expansions to both the gesture recognition and FSM control logic, as well as additional command messages to be passed between the controller and the robot interface. Utilizing the MATLAB code generation feature, in place of the hand-written FSM and gesture recognition code currently used, would significantly reduce the time required to add additional functionality to the system. Finally, as the Kinect is capable of tracking two full skeletons and was originally intended as a video gaming device, the controller software could be expanded to support two users simultaneously. From here, it is not much of a jump to a simple video game allowing the users to race two simulated, or possibly physical, robots around a suitable world model environment.

Team Member Roles

Ben had two main roles for this project. The first was the gesture recognition component, which included writing the initial program for the task, experimenting with it and the visualization of the skeleton using the Kinect SDK, and later refining it for the current higher success rates. The other task dealt with the FSM model. This includes everything done in Simulink such as recreating the gesture recognition process, and the Robot Controller model in Stateflow. Additionally, this eventually led to Ben completing the formal analysis of the FSM.

Charlie was responsible for writing the robot controller and Gazebo interface C++ code as well as setting up the Gazebo simulation environment. Messaging between the robot controller and the Gazebo interface also required writing and troubleshooting a simple TCP/UDP networking library. As previously discussed, the use of pre-existing models for the robot and the simulation environment significantly reduced the time required to setup and configure the Gazebo environment. This allowed for more time to be spent troubleshooting the Gazebo interface which turned out to be more time consuming than expected.

Key Concepts

One of the main ideas that we strove to apply in this project is modal behavior governed by a FSM coupled with formal analysis. The FSM aspect is easy to see throughout the project. We have the MATLAB Stateflow model along with an equivalent StateMachine.cpp class that both capture the FSM used to control our robot in Gazebo. What has not really been covered elsewhere is the formal analysis. While we did mention the problems we ran into when trying to use a specific Stateflow tool for the task, we reserved this for the space to discuss what we did about it. The first step was coming up with an LTL representation of what we believed to be the proper safety requirement for the model. This equation is $G\ (p \Rightarrow X\ (q\ U\ \neg p))$, where p = (Wall==1 ∨ Right_Sensor==1 ∨ Left_Sensor==1) and q = (Auto.TurnL ∨ Auto.TurnR ∨ Auto.Backward ∨ Manual.Avoid_Obstacle ∨ Manual.Stop.Stop). In words, this means that always when a sensor is triggered, the system immediately transitions to any of the safe states, and will remain in a safe state until such a time as no sensors are on. We do allow for transitions between the different safe states; while it is tough to represent in the Stateflow model, the idea for most of those states is to forcibly move the car away from the obstacle to turn off the appropriate sensor. The safe states are essentially those which are working to accomplish this, along with Stop which should just be a temporary state to transition to before moving on to one of the more productive options.

Actually verifying the idea above proved to be a bit of a problem. As stated, we do have some functionality in the Stateflow/Simulink model that compares the combined sensor values to whether or not we are safe as some amount of verification. To attempt to actually prove by formal analysis that our FSM obeys the LTL representation above, I will be referring to the cps.slx file in our attached code, and specifically the Robot Model component of that. When any of the sensor inputs happen, there are a few possible results. First, if the model is in any of Manual.Forward, Manual.Backward, or Manual.Stop, two different transitions could be made (because of the hierarchy of the model, the other normally available transitions will not be possible, and I will only be discussing the actually possible transitions). The first of these would be if the sensor triggered at the same time as a gesture indicating a change to autonomous mode is recognized. The result of this would be a transition to the Autonomous.Backward state, which is the initial step in autonomous mode in an attempt to allow the sensor to turn off. The other case would occur when a sensor is triggered and no change to autonomous mode input is present. Under this set of circumstances, the model will immediately transition to Manual.Avoid_Obstacle, which is a safe state designed to temporarily take control of the robot from the user and simply back up until the obstacle is no longer sensed. The next set of results occur when a sensor is triggered and we are currently in the Autonomous.Forward state. The two possibilities here are either transitioning to Manual.Stop if a gesture indicating manual mode is recognized at the same time, or transitioning to Autonomous.Backward otherwise. These transitions also bring us to a safe state.

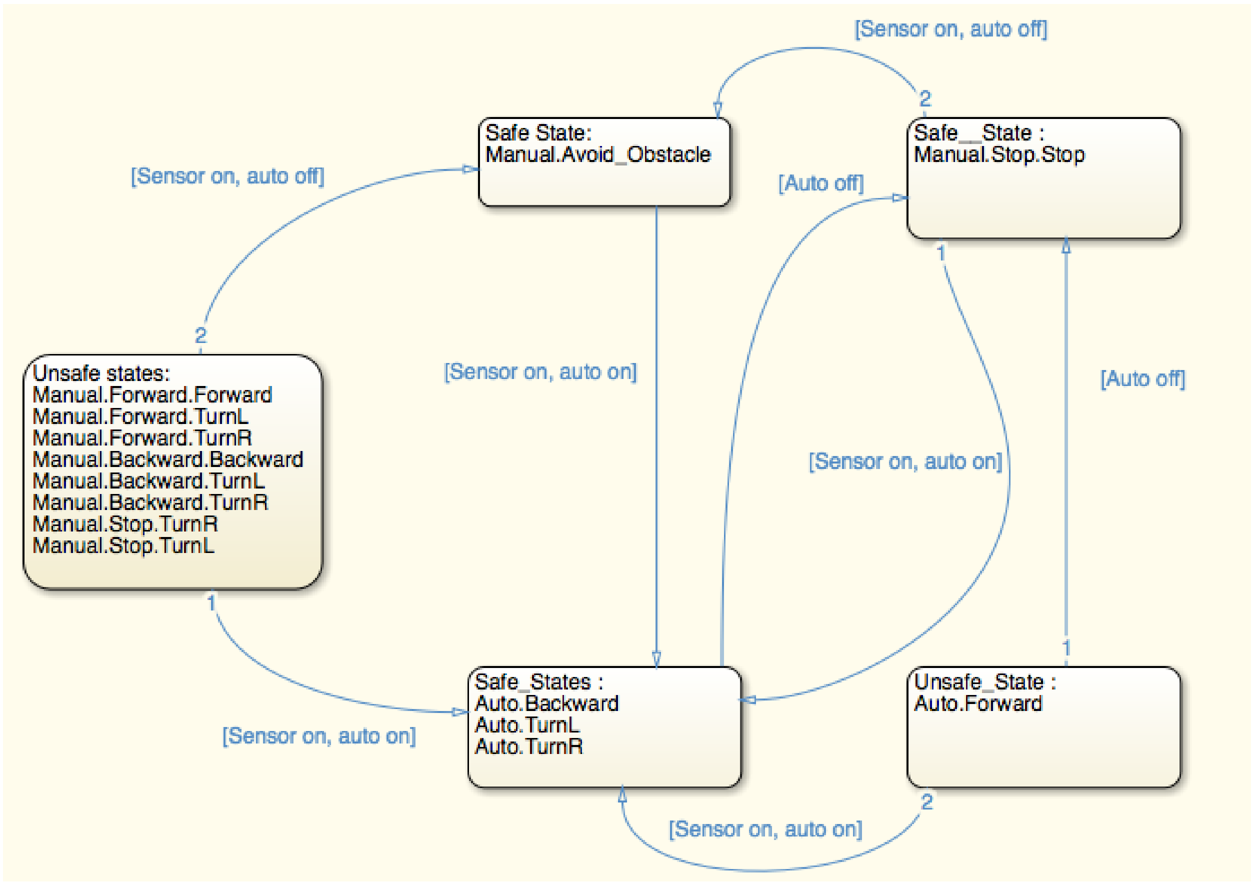

Figure 5. Simplified model of FSM to demonstrate behavior while a sensor is on

Finally, a third consideration is if we are already in a safe state. These transitions apply both when a sensor is first turned on as well as throughout the entire time that a sensor is enabled; thus, this will verify that we immediately transition to or remain in a safe state at the beginning while also remaining in a safe state until such a time that no sensors are on. Manual.Avoid_Obstacle has only one potentially enabled transition, which would be to Auto.Backward when an auto gesture is recognized. Manual.Stop.Stop would have that case and additionally would immediately transition to Manual.Avoid_Obstacle otherwise. The only enabled transitions for Auto.TurnR, Auto.TurnL, and Auto.Backward involve staying within those three states or transitioning to Manual.Stop.Stop upon receiving a command corresponding to the manual gesture. Figure 5 exhibits the simplest breakdown of the behavior of the model during a time where a sensor value is 1. Because the first transition occurs at the same time step as the sensor trigger, it will immediately transition to some safe state and then remain there until all sensors are 0 again. The only thing left to prove is that at some time after all sensors are cleared, the robot will leave the safe states and resume normal operation (assuming we are either in autonomous mode or that inputs are received). If the current state is Manual.Avoid_Obstacle, the transition to Manual.Stop.Stop is enabled as soon as all sensors are 0. From Manual.Stop.Stop, we can transition to any number of active states as soon as the proper gesture is recognized, such as turning in place, moving forward or backward, or switching to autonomous mode. Starting in Auto.Backward, the model would first wait for 50 ms and then transition to either Auto.TurnR if the left sensor is on or Auto.TurnL for all other situations. Both of these turning states will transition to the normal operation state Auto.Forward assuming no more sensors have been triggered. We have now shown all aspects of $G (p \Rightarrow X (q \, U \, \neg p))$ to be true.

The second of the main ideas for our project was simulation strategies. We applied several forms of simulation in order to complete the project efficiently without needing to use much in the way of physical hardware for the time being. One of these was the models created in Simulink and Stateflow. Simulink allows us to completely simulate the gesture recognition components being used. By changing the inputs that mimic the current x and/or y positions of each part of the body, we can get an accurate representation of exactly what outputs would occur. This method of testing by simulation was extremely useful, allowing us to troubleshoot the model eventually used to control the robot since they were functionally identical. After obtaining these outputs, the second part of the MATLAB simulation was available to see what would happen in the Stateflow diagram. Again, the Stateflow model was set to exactly mimic the state machine being used by the robot, allowing for simulation results of significance. These simulations enabled us to test that our model satisfied the safety and liveness requirements that we had set in a more controllable way than actually performing the gestures with the Kinect sensor. The other main type of simulation we were using is the Gazebo environment. The MATLAB models were pretty much necessary from a testing standpoint, allowing us to try out things that would have taken a much longer time with physical constraints. Similarly, by using a Gazebo environment and simulated robot as opposed to a real robot, we were able to get a lot more done for the time being and test more than we would have been able to otherwise. We had hoped to meet our stretch goal to work with a real robot, and believe that the work we have done is easily adaptable for that task. However, a simulated environment and robot was our best option to complete everything in a timely manner. We had already done some previous work with Gazebo, so choosing that as the platform for the simulation was a pretty simple decision.

Our project also touched on two additional topics from the course including real-time networking and concurrency. All sensor data and robot commands were sent over a typical wifi network, meaning that this network would need to have a limit on maximum latency in order for the system to function properly and prevent a crash. If the iRobot Create were approaching a ledge at maximum speed (about 500 mm/s), it would have approximately 300 ms between the forward sensors being triggered and the robot falling off the ledge. This means that the round trip time of sensor data being received and the stop command being sent has an upper limit of 300 ms, with a more realistic limit of 200 ms or less to allow for the robot to have sufficient time to come to a halt. The local-area networks used in our testing provided typical latencies of less than 10 ms, well within this limit. However, this limitation would become significant if the control system were used over a wide-area network where the overall transport latency could approach or exceed the upper limit on response time. Additionally, the Robot Controller was a multi-threaded program, and as such used many of the concurrency topics discussed in class. In particular, the proper methods for sharing data between threads were used extensively in our system

Feedback

Many of the topics covered in class were useful to us throughout the course of this project. A major component of our project was the finite state machine, so the chapter on discrete dynamics was a good overview of FSMs. It assisted us in determining the proper guard conditions on transitions, and offered information on the stuttering reactions which are common for our model. Looking at our Simulink/Stateflow models, it is clear that we were able to apply many of the concepts from chapter 5. We composed states for Autonomous and Manual mode together, each of which have many states contained within. This is an example of hierarchical FSM's, where we refine one state as another state machine - because of our understanding of this lesson, we were able to implement our control logic FSM in this way to make our model far less cluttered and easier to comprehend. Additionally, a few of the more recent topics were of use when it came to analyzing the FSM. For example, we were able to use the ideas of temporal logic to come up with a LTL formula for safe operation of the model. We also incorporate some ideas from Chapter 15, such as liveness properties - assuming inputs to the system are received, the robot should eventually leave the stop state and begin to move in some way. Finally, the multitasking topics covered in class were useful due to the multi-threaded nature of our robot controller software. The threads utilized mutexes to ensure that data consistency was maintained when sharing data between threads [3].

While many of the topics covered in class were utilized, there are other topics that could have been covered which would have been very beneficial to the development of our project. The formal analysis methods covered in class would have been of more use if more practical means of performing such analysis had been discussed in detail. Many of the mathematical models presented for formal analysis are very difficult to apply to systems more complex than the in-class examples. A discussion of the practical and commonly available software tools for performing such analysis would have been particularly useful.

Conclusion and Related Work

Our project successfully implements a robot control system wherein a simulated iRobot Create is controlled by the user via simple gestures. In its current state, the system supports a small set of gestures and has only been tested with a basic simulated wheeled robot. However, the model based design of the system allows for straightforward expansion to more complex robots with additional functionality. We are not the first to try to implement gesture recognition using Kinect, nor the first to apply this to some form of robotics. We found a few examples of related projects within a short time of searching. IEEE Spectrum lists some recent (2011) ideas that use the Kinect for robotics tasks, which includes a video of a Roomba being controlled by hand motions similar to those used to actually push around a vacuum [4]. Other uses for this kind of technology on the site are teleoperation of a robot (with arms, etc.) through gestures and even performing surgery using a robotic arm controlled by gestures [4]. While our project is relatively simple due to the time constraints of the course, it is clear that ideas such as these are applicable to many tasks in the world today, and the future work that could be done along this line has a lot of potential.

Acknowledgements

We would be remiss to not recognize a first source of help when it comes to the gesture recognition process using Kinect. A Google search for Kinect gesture recognition will quickly turn towards a good source of information named Vangos Pterneas. This gentleman breaks down a gesture into its distinct parts and provides visualization of exactly what the Kinect is seeing and how to translate that into code. He had even provided an open-source library containing nine implemented gestures among other tools for Kinect applications. Ultimately, we wrote our own code from scratch, using Mr. Pterneas' main website as an occasional reference while never getting into the library. This was partly because he was using a .NET framework while the two of us preferred to use C++ due to previous experience with the language, and partly because we thought it would be a more interesting and challenging problem to come up with our own approach [5].

We also would like to thank Microsoft for making the Kinect SDK freely available. The SDK, in particular the included example programs, turned out to be a quite useful piece of the puzzle, as we were able to use a demo known as SkeletonBasics (selecting the D2D version as it was written in C++) for help in a few key areas. First, the complete demo itself would display the skeleton as recognized by the Kinect on screen, which allowed us to verify how each body part was represented and also to test and troubleshoot gesture recognition later on. Additionally, the code for the demo introduced us to the best means to access the data being processed by the Kinect [6].

References

[1] *Hand Raised For Stop*. 2016. Web.

[2] *Elbow Bends*. 2017. Web.

[3] E. A. Lee and S. A. Seshia, Introduction to Embedded Systems - A Cyber-Physical Systems Approach, Second Edition, MIT Press, 2017.

[4] E. Ackerman. (2011, March 7). *Top 10 Robotic Kinect Hacks*. [Online]. Available: http://spectrum.ieee.org/automaton/robotics/diy/top-10-robotic-kinect-hacks

[5] V. Pterneas. (2014, January 27). *Implementing Kinect Gestures*. [Online]. Available: http://pterneas.com/2014/01/27/implementing-kinect-gestures/

[6] Microsoft. (2017). *Kinect for Windows SDK v1.8*. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=40278